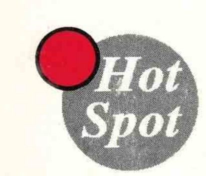
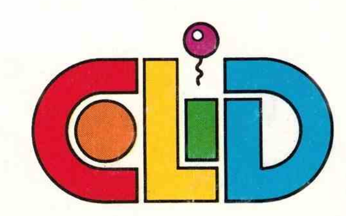
Litzkendorf

HotSpot

GFA BASIC für MSDOS





Printed in GERMANY ©

Copyright 1992 by	COLID Verlag GmbH Schmiedestrasse 18 3000 HANNOVER 1
HotSpot GFA-BASIC für MSDOS	
1.Auflage	1992 - Hannover
ISBN	3-9802925-0-9
Umschlaggestaltung	
Grafik und Layout	Uwe Litzkendorf
Textverarbeitung	GFA-BASIC Editor
DeskTopPublishing	PC - AldusPageMaker4.0
Vorlagenausdruck	NEC Silentwriter 2 S60
Druck und Verarbeitung	Schlütersche Verlagsanstalt und Druckerei, Hannover



Danksagung

Anfang 2019 begann ich damit, alles an Atari-Hardware zu erwerben, was ich bereits in den 80er/90er-Jahren schon einmal besaß. So auch Bücher, die sich speziell mit dem Thema Programmierung beschäftigen. Ich stellte schnell fest, dass dieses Vorhaben heutzutage schwieriger werden könnte als der Kauf eines Atari ST. Für eine gebrauchte Ausgabe von „Das große GFA Basic 3.5 Buch“ zahlte ich mehr als den damaligen Neupreis! Mein Wunsch war aber, genau dieses Buch wieder in meiner Sammlung zu sehen. Da das Buch sehr gut erhalten war, mochte ich es auch so belassen und scannte es daher ein. Somit konnte ich nun bequem digital darin blättern, nach Begriffen suchen und Lesezeichen erstellen. Später entstand allerdings der Gedanke, dass diese digitale Version doch eigentlich der Allgemeinheit zur Verfügung gestellt werden sollte. Eine Anfrage zur Genehmigung beim Data Becker Verlag war allerdings nicht mehr möglich, da der Verlag seinen Geschäftsbetrieb zum 31.03.2014 aufgegeben hatte und es auch keine Rechtsnachfolger gibt. Somit machte ich mich auf die Suche nach dem Autor, Uwe Litzkendorf, und fand ihn auch. Mit großer Freude durfte ich feststellen, dass er meine Meinung teilte. Und mehr noch, Uwe machte den Vorschlag, dass ich es nicht nur bei diesem einem Buch belassen sollte, sondern auch weitere seiner geschriebenen Fachbücher zum Thema GFA-Basic digitalisieren dürfte. Ich war begeistert und erklärte mich gerne bereit dieses Projekt durchzuführen. So entstand aus einer einfachen E-Mail Anfrage eine wunderbare Möglichkeit den Inhalt dieser Bücher digital für die Zukunft zu erhalten und allen Interessierten kostenfrei zur Verfügung stellen zu können. Dafür möchte mich bei Uwe noch einmal sehr herzlich bedanken.

Thomas Werner

Bearbeitungsstand: 03. Mai 2020

Neues Vorwort zur Digitalisierung

Es hat lange gedauert! Aber nun hat sich Thomas Werner die viele Arbeit gemacht, den größten Teil meiner Bücher nach, zum Teil über dreißig, Jahren in interaktiver PDF-Form zu digitalisieren und als Public Domain kostenlos online zur Verfügung zu stellen. Nachdem meine Bücher einen langen Dornröschenschlaf hinter sich haben, sollen sie und das immense darin festgeschriebene Wissen jetzt wieder zum nützlichen Leben erweckt werden :O)

Meiner Ansicht nach ist die objektorientierte Programmierung nicht für die Massenanwendung geeignet. Nach meiner Berechnung sind nur ca. 15 Prozent der Bevölkerung in der Lage, mit der OOP sinnvolle Erfolge zu erzielen. Um aber "massenfähig" zu sein, muss eine Programmiersprache mindestens 80 Prozent der Bevölkerung erreichen, damit sich Lehrer und Schüler auch außerhalb von IT-Leistungskursen über die Softwareentwicklung so verständigen können, dass der eine – vermittelbar und auch prüfbar – wenigstens "einigermaßen" versteht, was der andere meint. Andererseits "reißt unweigerlich der Faden" zwischen Ausbildern und Lernenden. Daher ist es auch kein Wunder, dass sich die prozeduralen und damit auch leicht anwendbaren Basic-Programmiersprachen, wie z.B. das sehr populäre GFA-Basic, einer gewissen Renaissance erfreuen.

Diese Tendenz werde ich natürlich als ehemals populärer Bestseller-Autor nach all meinen Möglichkeiten tatkräftig unterstützen. Ich verfüge über mehr als 4000 Seiten umfassenden, ausführlichen und auch unter den wachsamen Augen der Öffentlichkeit profund geprüften Software-Wissens.

Dieses Wissen ist auch in der heutigen Zeit absolut nicht überflüssig und veraltet, sondern bildet auch heute noch die Basis für algorithmisches Grundlagenwissen.

Aber damit nicht genug: ich habe zudem beschlossen, eine neue Programmiersprache namens "QS!X©" zu entwickeln. Sie wird sich in vielen Punkten an einfachem Standard-Basic anlehnen. Auf der weltweit überall auf allen Betriebssystemen und in jedem Standardbrowser verfügbaren – und damit 100% cross-kompatiblen – Plattform von HTML5/Canvas wird "QS!X©" als Open Source-Version (ähnlich LINUX) verfügbar sein. Nähere Informationen dazu finden Sie unter:

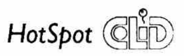
http://www.litzkendorf.net/invitation_info_d.pdf

Damit ist auch der "Klasse für die Masse"-Philosophie von Frank Ostrowski (dem GFA-Basic-Vater) Rechnung getragen. Wenn denn alles so funktioniert, wie ich es mir vorstelle, wird die weise, sanfte und erzfreundlich geduldige und bescheidene Denkart von Frank Ostrowski auch Jahre nach seinem (viel zu frühen) Ableben noch weltweit merkliche Wirkung tragen! Er bildet dann verdientermaßen – zumindest im IT-Business – die philosophische Grundlage für eine Art "Weltsprache"! Das würde ihm sicher sehr gefallen!

In treuem Gedenken an einen wirklich großen Mann, mit dem ich das Vergnügen hatte, teil- und zeitweise recht eng und vertraut zusammen zu arbeiten und dem mit "QS!X©" auch ein digitales – und verdientes – Denkmal gesetzt wird!

Uwe Litzkendorf
Hildesheim, im Mai 2020

INHALTSVERZEICHNIS



Vorwort

Insider sind sicherlich nicht besonders verwundert, daß nun nach ca. 5 Jahren GFA-BASIC-Euphorie auf dem ATARI-ST auch die entsprechenden MSDOS-Versionen, sowie auch eine WINDOWS-Version dieser phantastischen Programmiersprache erhältlich ist. Wer die Entwicklung dieses BASIC's in den letzten Jahren verfolgt hat, weiß, wie sehr sich diese - anfänglich noch recht bescheidene Sprache - vom *'Hoppla, da bin ich...'*-Typ zu einem absolut ernstzunehmenden und eigenständigen Entwicklungs-System gemausert hat. Das führte dazu, daß selbst angesehene Software-Häuser und C-verschworene Profi-Programmierer ein (oft auch zwei) Auge(n) auf diese äußerst komfortable und unkomplizierte Möglichkeit der Programmentwicklung warfen.

Es stellte sich heraus, daß es in kaum einer anderen Sprache für jedermann so leicht möglich sein sollte, seinen Programmen ein professionelles, komfortables und zugleich auch schnelles Outfit zu verleihen. Dem eingefleischten C-Programmierer bleibt anhand des perfekten GFA-BASIC- Compilers dann noch die Möglichkeit, zeitintensive oder z.B. rekursive Programmteile (kann GFA-BASIC auch, nur nicht ganz so schnell) als C-Module in das BASIC-Programm einzulinken.

Wer mein *'Großes GFA-BASIC-Buch'* zum Atari ST kennt, wird ahnen können, daß ich eine begeisterter Fan des Assembler-Genius' und GFA-BASIC-Vaters Frank Ostrowski bin. Auch zum GFA-BASIC unter MSDOS und MS-WINDOWS zolle ich ihm hiermit meine uneingeschränkte Hochachtung.

Ich kenne den Werdegang einer GFA-BASIC-Version aus nächster Nähe und weiß aus guter Erfahrung, daß die ersten Vorab-Versionen noch hier und da Mängel und logische Fehler beinhalten. Doch ich weiß auch, daß diese Fehler nach kurzer Zeit gründlich ausgemerzt sein werden und immer wieder neue, mächtige Befehle und Funktionen zu einem erheblichen 'Nutzen-Wachstum' führen. Der kundenfreundliche Update-Service der Firma GFA-Systemtechnik gewährleistet zudem die Möglichkeit, neueste BASIC-Versionen für jedermann erschwinglich zu machen, sodaß in kurzer Zeit ganze Heerscharen enthusiastischer Programmierer sich dieser legendären Sprache bemächtigen werden.

Erste Anzeichen deuten bereits heute darauf hin, daß schon kurz nach Markteinführung ein großer Teil der GFA-BASIC-Routiniers am Werke ist, um den gewaltigen Software-Bestand, der auf **ATARI ST** oder **COMMODORE AMIGA** unter **GFA-BASIC** produziert wurde, auch unter **MSDOS** und **WINDOWS** verfügbar zu machen. Nach anfänglichen Schwierigkeiten überraschte mich die doch weitgehende Kompatibilität zwischen dem ATARI-

GFA-BASIC und dem GFA-BASIC für MSDOS. Es bleiben zwar - wie sollte es bei zwei so unterschiedlichen Systemen auch anders sein - unübersehbare Probleme bei der Anpassung, aber dieser Mangel wird sicherlich in recht kurzer Zeit dadurch behoben sein, daß Konverter erhältlich sein werden, die diese Übertragung dann erleichtern.

Aus all diesen vorgenannten Gründen haben wir uns das Ziel gesetzt, Ihnen mit diesem '**COLID**-*HotSpot*' in kürzestmöglicher Frist einen kompetenten und umfassenden Schnell-Einstieg zu bieten, damit Sie 'am Ball bleiben'. Dieser '*HotSpot*' stellt eine umfassende Kurzbeschreibung aller GFA-BASIC-Befehle unter MSDOS zur Verfügung.

Bei diesem Schnell-Einstieg soll es allerdings nicht bleiben. Wir werden circa zum Mai 92 ein GFA-BASIC-Buch der Premium-Klasse vorstellen. *'Das große GFA-BASIC-Buch zum Atari ST'* ist Dank des wohlwollenden Interesses meiner Leser zu einem der meistverkauften BASIC-Bücher geworden (Gesamtauflage: 65.000 Exemplare). Dieses überaus erfolgreiche Nachschlagewerk wird nun in seinem gesamten Umfang an die MSDOS- und WINDOWS-Versionen, sowie später auch an die anderen Versionen des GFA-BASICs (z.B. UNIX) angepaßt und dann jeweils unter dem Titel *'Das PREMIUM-Buch zum GFA-BASIC für...'* im COLID-Verlag erscheinen.

Alle systemspezifischen Anpassungen werden dabei berücksichtigt sein und eine Fülle ausgesuchter und nützlicher Routinen zum direkten Einsatz in Ihren eigenen Programmen zur Verfügung stehen. Auch mit dem Thema 'Programmportierung vom ATARI-GFA-BASIC zum GFA-BASIC für MSDOS' werden wir uns in diesem Buch näher befassen. Daß dieses Buch dann auch über umfassende und sinnvolle Index-, Syntax- und Befehlslisten, sowie Themenübersichten und aufschlußreiche Querverweise verfügen wird, ist für meine bisherigen Leser schon zur Selbstverständlichkeit geworden.

Lassen Sie sich also angenehm überraschen!

Uwe Litzkendorf im Januar 1992

1. HOTSPOT GFA-BASIC FÜR MSDOS

1.1. ZUM ALLGEMEINEN VERSTÄNDNIS

Das Anliegen dieses kleinen Buches ist es, die mehr als 530 Befehle, Funktionen und Operatoren und Variablen des GFA-BASICs für MSDOS nach thematischen Schwerpunkten zu ordnen, um so das Auffinden der gesuchten Befehlsbeschreibungen nach problemorientierten Gesichtspunkten zu ermöglichen bzw. zu erleichtern.

Wir gehen dabei davon aus, daß Sie als Leser dieses *HotSpot*'s mit den grundsätzlichen Verfahrensweisen bzw. den Möglichkeiten der Anwendungs der meisten Befehle von anderen BASIC-Dialekten her schon einigermaßen vertraut sind. Weiterführenden Dokumentationen zu den Befehlen - wie umfangreiche Beispiel-Programme zu den einzelnen Befehlen oder Tips und Tricks - werden Sie hier - bis auf einige Bonbons (siehe z.B. **STR$()**) - vergeblich suchen.

Dieses Buch ist so angelegt, daß Sie neben dem Inhaltsverzeichnis einerseits über den Stichwort-Index (s. ANHANG) und andererseits über die Befehlsliste (s. ANHANG), sowie auch über die assoziative 'THEMENÜBERSICHT' (s.ANHANG) zum Ziel gelangen. Innerhalb der Kapitel wurde versucht, weitestgehend die alphabetische Reihenfolge der Befehle und Funktionen zu berücksichtigen. In einigen Fällen konnte dieses Schema jedoch nicht eingehalten werden. Richten Sie sich dann bitte nach der eben genannten 'THEMENÜBERSICHT', wo die Befehlsbeschreibungen möglichst nach deren thematischer Zusammengehörigkeit gegliedert wurden.

Neben einer klaren Gliederung wird die Effizienz dieses *HotSpot's* durch einige Vorgaben zur einheitlichen Darstellung gewährleistet, die im gesamten Text Anwendung finden:

< > Wird bei einer Befehlsbeschreibung auf bestimmte Tasten verwiesen, wird ihr Name zur besseren Kenntlichmachung in spitzen Klammern angegeben (z.B. <Shift>, <A>, <Return> oder <Undo>). Da diese Klammern nicht inner halb von Listings oder Syntaxbeschreibungen auftauchen, sind Sie auch leicht von *'größer als / kleiner als'*- bzw. *'Bitshift'*-Pfeilen zu unterscheiden.

[ ] Bei Befehlen, deren Syntax variabel ist, wird ein optionaler Befehlsteil in eckigen Klammern angegeben. Dies bedeutet, daß die Angabe (z.B. [;,'] oder [,*Länge*])

nur dann im Befehl angegeben werden muß, wenn die damit verbundene Option genutzt werden soll.

{ } Viele GFA-BASIC-Befehle können als Abkürzung angegeben werden. Der Interpreter erweitert diese dann selbstständig auf die richtige Form. Sollte zu einem Befehl eine Kurzschreibweise existieren, ist diese in der Titelzeile und in der Quick-Referenz innerhalb von geschweiften Klammern angegeben (z.B. { SYS } oder { RET }). Diese geschweiften Klammern werden auch von einigen BASIC-Befehlen (Speicherzugriffe wie **CHAR{}**, **BYTE{}** etc.) verwendet. Die Verwechslungsgefahr mit den hier gemeinten Abkürzungsklammern ist jedoch gering, da diese aus schließlich in der Kopfzeile der jeweilige Befehlsbeschreibung verwendet wurden.

... Soll eine Folge von Anweisungen innerhalb von Befehlen verdeutlicht werden, geschieht dies anhand einer Punktlinie (z.B. **FOR ... NEXT**).

Grundsätzlich sind in der Syntax-Beschreibung alle Befehlsnamen in Großbuchstaben, alle Variablen, Parameter und Strings in normaler Schreibweise dargestellt (z.B. **OPENW #***Handle*).

Bei den Parameterangaben wurden weitgehend einheitliche Bezeichnungen verwendet:

Adresse = Speicheradressen werden unter MSDOS grund sätzlich im Wort-Format als Segment-Startadresse und Byte-Offset durch einen Doppelpunkt getrennt ange geben (Segmenwort:Offsetwort) z.B. $A000:800 oder 5464:20*40.

Anz = Anzahl

Arg = Funktionsargument

Back = Rückgabedaten bei Funktionen

Expr/Expr$ = numerischer/alphanumerischer Ausdruck

Feld() = beliebige Feldbezeichnung

Index = Index von Feldelementen

Kanal = Datei-Identifikator

Nummer = GFA-Fensternummer

Quell$	=	beliebiger Textausdruck, Zeichenkette oder Stringvariable, die einer Stringfunktion als Ausgangstext dient.
Quell	=	Startadresse (Segment:Offset) des Quellbereichs einer Speicheroperation (s.oben unter *'Adresse'*)
Quell()	=	Fließkommafeld, das bei MATxxx-Befehlen als Quellfeld dient
Start	=	meistens String-Startposition für stringbezogene Operationen
Text$/"Text"	=	beliebige Zeichenkette, die in den meisten Fällen auch als Stringvariable übergeben werden kann
Var/Var$	=	beliebiger Variablenname
Xpos/Ypos	=	Bildschirmkoordinaten
Ziel	=	Startadresse (Segment:Offset) des Zielbereichs einer Speicheroperation (s.oben unter *'Adresse'*)

Bei *'Dateiname', 'Programmname' und 'Ordner'* ist davon auszugehen, daß ein evtl. erforderlicher Suchpfad in den Namen einzubinden ist.

Bei einigen Befehlen und Funktionen kam es darauf an, zwischen vorzeichenbehafteten und vorzeichenlosen Daten und Werten zu unterscheiden. Um diese beiden langen Worte nicht jedesmal voll ausschreiben zu müssen, wurde häufig statt 'vorzeichenbehaftet' die Bezeichnung 'signed' und statt 'vorzeichenlos' die Bezeichnung 'absolut' verwendet.

Unter dem Begriff *'Ausdruck'* (s.o. *'Expr'*) wird hier eine beliebige Zusammenstellung von Konstanten, Formeln, Texten, Funktionen und Variablen verstanden, die zusammen ein Ergebnis liefern:

```
A%=B%+((234^2/4.7)*12.95*C%)^2.1317+@Func(Abc%)
   |                                          |
    ---- z.B. numerischer Ausdruck = Expr ----
```

oder

```
A$=''Text''+STR$(A%*B%)+SPACE$(10)+@Func$(Abc$)+B$
   |                                             |
    --------   z.B. Text-Ausdruck = Expr$ --------
```

In den Beschreibungen von Funktionen wird nicht explizit angege-

ben, daß die Ergebnisse **aller** (auch selbstdefinierter) Funktionen auf verschiedene Weisen ausgewertet werden können. Funktionsaufrufe stehen immer stellvertretend für einen Wert oder String, den diese Funktion liefert. Sie können deshalb also wie jeder beliebige Wert, String oder Ausdruck in Programmzeilen verwendet und eingesetzt werden.

Einsatz von selbstdefinierte Funktionen:

```
Var%=@Func              (als Zuweisung)
PRINT @Func             (als Ausgabe)
IF @Func=X THEN ...     (als Abfrage)
VOID @Func              (als Dummyaufruf)
```

Einsatz von BASIC-Funktionen (hier: FRE)

```
Var%=FRE(0)             (als Zuweisung)
PRINT FRE(0)            (als Ausgabe)
IF FRE(0)=X THEN ...    (als Abfrage)
VOID FRE(0)             (als Dummyaufruf)
```

In den Syntaxzeilen von Funktionen wird in diesem Buch zur Verdeutlichung die Zuweisungsvariante ('`Var=Funktion()`') verwendet.

Notizen:

1.2. DER EDITOR

Nachdem Sie das GFA-BASIC gestartet haben, erscheint ein fast leerer Bildschirm, an dessen oberen und unteren Rand je eine andersfarbige Zeile angeordnet ist. Dies ist die Programmeingabe-Oberfläche des GFA BASIC: der Editor. Die obere Zeile enthält mehrere Menütitel, hinter denen sich die eigentlichen Editorfunktionen verbergen (s.u.: 'DAS EDITOR-MENÜ'). In der unteren Zeile befindet sich die Angabe der aktuellen Zeilen- und Spaltennummer, sowie eine Statuszeile, bzw. die Angabe des aktuellen Dateipfades.

Sofern sie (ratsamer Weise) über eine Maus verfügen und der dazugehörige Treiber installiert ist, erscheint ein dunkles Rechteck, das mit der Maus über den Bildschirm bewegt werden kann. Durch einen Links-Mausklick kann mit diesem Zeiger der Cursor (=blinkender Unterstrich) an eine beliebige Position innerhalb des Programmtextes (soweit vorhanden) bewegt werden. Befindet sich der Mauszeiger auf einer der obersten oder untersten vier angezeigten Programmzeilen, so kann das Listing durch Rechts-Mausklick in die entsprechende Richtung gescrollt werden.

Der GFA-BASIC-Programmeditor funktioniert im wesentlichen wie eine 'normale' Textverarbeitung. Bei genauerer Betrachtung fallen jedoch schnell einige Merkmale auf, die ihn doch weit davon unterscheiden. Der wesentlichste Unterschied ist, das eine geschriebene Programmzeile nur dann verlassen werden kann, wenn der Editor einen Syntax-Check durchgeführt hat und die Zeile als syntaktisch richtig erkennt. Ist sie fehlerhaft, bleibt der Cursor in der Zeile stehen und es erscheint in der Fußzeile der Hinweis *'Syntax Error'*.

Diese Eigenschaft ist einer der wichtigsten strukturellen Vorteile, die GFA-BASIC von anderen BASIC-Dialekten unterscheidet. Bis zu 90 Prozent aller möglichen Fehler können so schon bei der Programmerstellung vermieden werden, ohne daß evtl. Runtime-Errors erst durch den Programm- oder Compilerstart gemeldet werden und mühselig nachkorrigiert werden müssen. Von den bisher gängigen BASIC-Dialekten im MSDDOS-Bereich ist man einen recht großen Debug-Aufwand gewöhnt. Erfahrene Profiprogrammierer bestätigen immer wieder, daß ein solcher Aufwand in GFA-BASIC kaum notwendig wird, weil dieser Syntax-Check den weitaus größten Teil an Fehlerquellen schon bei der Programmentwicklung unterbindet.

GFA BASIC verzichtet - wie die meisten modernen BASIC-Dialekte - gänzlich auf Zeilennummern. Da diese Sprache prozedural orientiert ist und in der 'äußersten Not' ein **GOTO** auch auf ein

beliebiges Sprunglabel derselben Struktur (auf der Hauptebene oder in PROCEDUREs und FUNCTIONs) gerichtet werden kann, ergibt sich auch keine Notwendigkeit dazu. Innerhalb des Editors kann die aktuelle Programmzeile bei Bedarf in der Fußzeile abgelesen werden.

Eine weitere unübersehbare Eigenschaft besteht darin, daß der GFA-BASIC-Editor selbstständig eine optische Programmstruktur generiert, wobei alle struktureinleitenden Befehle (z.B. **IF, PROCEDURE, REPEAT, DO** etc.) bewirken, daß die folgenden Programmzeilem um zwei Zeichen nach rechts versetzt werden und alle strukturschließenden Befehle (z.B. **ENDIF, RETURN, UNTIL, LOOP** etc.) um zwei Zeichen nach links versetzt werden und dabei nachfolgende Zeilen mitziehen. Dies gewährleistet einen schnellen Überblick über die allgemeine Programmstruktur und erleichtert die Suche nach logischen Fehlern ganz erheblich.

Die nächste Besonderheit wird dann erkennbar, wenn man - bei den meisten Befehlen - statt des kompletten Befehlsnamens die dafür vorgesehene Abkürzung verwendet (z.B. 'P' statt **'PRINT'**). Der Editor ergänzt diese Abkürzungen automatisch, sodaß hierdurch bei der Programmerstellung viel Tipparbeit eingespart werden kann. Hier im Buch sind die Abkürzungen in den Kopfzeilen der jeweiligen Befehlsbeschreibung zwischen zwei geschweiften Klammern (z.B.: ABKÜRZUNG **{ ABK }**) aufgeführt.

Aus meiner langjährigen Erfahrung weiß ich, daß gerade diese Fähigkeiten des Editors - neben der überaus hohen Ausführungsgeschwindigkeit des Interpreters - wesentlich zu Zeiteinsparungen beitragen. Wer schon einmal ein umfangreiches kommerzielles Programm geschrieben hat, wird wissen, daß die Zeitspanne eines halben oder gar eines ganzen Jahres für die Erstellung eines solchen Programmes keinen besonders großen Aufwand darstellt. Mit diesen Zeiträumen (und noch mehr) ist bei größeren Projekten ohne weiteres zu rechnen. Wenn durch die ausgefeilten Editorfunktionen - dazu zählen auch die unten aufgeführten Tastaturfunktionen - nur 10 Prozent (erfahrungsgemäß sind es mehr!) der Zeit eingespart werden könnten, so wäre das - auf ein Jahr bezogen - **eine Einsparung von ca. einem Monat (!!).**

Die Raffinesse und Mächtigkeit vieler GFA-BASIC-Befehle (Fenster-, Menu- und Dialogbefehle, sowie z.B. **RC_INTERSECT, BMOVE, MEMXOR, EMSxxx** etc.) spart zusätzlich beträchtliche Zeit. Daß solche Befehle natürlich auch die Effizienz und Qualität des Programmes steigern, ist dabei selbstverständlich.

1.3. DAS EDITOR - MENÜ

Die im folgenden beschriebenen Editor-Menüs ermöglichen eine Bedienung sowohl per Maus (sofern vorhanden), als auch per Cursortasten. Der Markierungsbalken kann durch die *'Auf-'* und *'Abwärts-Cursortaste'* in die gewünschte Richtung ge*'scrollt'* werden. Die Einträge der Menüs sind weitestgehend mit Tastaturkürzeln wählbar. Um den entsprechenden Eintrag auszuwählen, drücken Sie bitte den angegebenen *'Hotkey'* oder bewegen den Markierungsbalken auf den Eintrag und drücken die <Return>-Taste. Mit der Maus haben Sie die Möglichkeit, diese auf den gewünschten Editor-Menüpunkt bzw. den jeweiligen Eintrag zu fahren und mit der linken Maustaste zu aktivieren.

'File' - Menü

(Datei-Operationen) öffnen: **<F1> oder <Alt>+<F>**

Load **<L>**: Token-Programm (Default *'.GFA'*) laden. Ein hiermit geladenes Programm überschreibt das aktuelle Programm vollständig. Gfls. vorher das aktuelle Programm sichern (vgl. **LOAD**).

Save **<S>**: Programm im GFA-*Tokencode* speichern. Das aktuelle Programm wird unter dem eingestellten Pfad (s. Fußzeile) und dem zuletzt gültigen Dateinamen ohne Nachfrage gespeichert. Existiert für dieses Programm noch kein Dateiname, wird es unter *'TEST.GFA'* gesichert.

Save As **<A>**: Programm im GFA-*Tokencode* speichern. Es erscheint eine FILESELECT-Box, in welcher die Eingabe eines Dateinamens im aktuellen Pfad (s.Fußzeile) erwartet wird. Hiermit (oder mit **SAVE**-Befehl) abgespeicherte GFA-BASIC-Programme erhalten - sofern keine andere vorgegeben wurde - automatisch die Extension *'.GFA'*.

Merge **<M>**: Programm im ASCII-Textformat laden. Ein Programm mit der Default-Extension *".LST"* wird in das aktuelle Programm ab Cursor-Position eingefügt. *'Gemergte'* Zeilen, die nicht korrekt interpretiert werden können, werden auskommentiert und mit dem Symbol =>> markiert.

Write **<W>**: Das aktuelle Programm wird als *ASCII-Textdatei* - falls nichts anderes vorgegeben wurde - mit der Extension *".LST"* im aktuellen Pfad gespeichert speichern. Es erscheint vorher eine **FILESELECT**-Box, in welcher der gewünschte Name einzutragen ist.

Print **<P>** : Nach Bestätigung einer Abfrage wird das aktuelle Programm gfls. auf dem Drucker ausgegeben.

New **<N>**: Das aktuelle Programm wird nach einer Sicherheitsabfrage gfls. aus dem Speicher gelöscht.

Ver X.XX XX : Aktuelle Versionsnummer, hat sonst keine Bedeutung.

Check **<C>** : Ist dieser Menüpunkt *'abgehakt'*, so wird dadurch bestimmt, daß bei Einführung neuer Variablen-, Label-, Prozedur- oder Funktionsnamen eine Meldung ausgegeben wird. Anhand dieser Meldung kann dann entschieden werden, ob der neue Name als Bestandteil in das Programm übernommen wird oder nicht. Falls nicht, wird die Eingabe als Syntax-Error abgewiesen. Bei Editor-Start ist diese Funktion ausgeschaltet.

Lower Names **<O>** : Ist dieser Menüpunkt *'abgehakt'*, kann dadurch entschieden werden, ob bei Eingabe von Variablen- Funktions- Label- oder Prozedurnamen alle großgeschriebenen Buchstaben automatisch in die Kleinschreibung konvertiert werden. Ist dies nicht der Fall, so werden sowohl groß als auch kleingeschriebene Namen vom Editor akzeptiert. Wurde allerdings an einer Programmstelle z.B. ein Variablenname groß oder gemischt geschrieben und derselbe Name wird erneut in kleiner oder sonst anderer Schreibweise eingegeben, so übernehmen alle gleichlautenden Namen intern automatisch die zuletzt verwendete Schreibweise.

Font **<F>** : Umschaltung der Editor-Schriftgröße (23 oder 48 Zeilen)

Exit **<X>** : Der GFA-BASIC-Editor wird nach einer Sicherheitsabfrage gfls. in Richtung DOS verlassen

'Search' - Menü

(suchen/ersetzen) öffnen: **<F2> oder <Alt>+<S>**

Find **<F>** : Ausdruck ab Cursorposition suchen. Es erscheint in der Fußzeile eine Aufforderung zur Eingabe eines Suchstrings. Ein gfls. vorher schon eingegebener String wird voreingestellt. Die Eingabezeile kann durch <Esc> gelöscht werden. Durch <Return> im Anschluß an die String-Eingabe wird dann die Suche ausgelöst (**<Esc>**, dann **<Return>** = Abbruch). Weiteres Suchen erfolgt dann über **<Strg>+<F>** bzw. **<Ctrl>+<F>**.

Strings innerhalb *'versteckter'* Prozeduren werden nicht gefunden. Wird der Ausdruck gefunden, wird die entsprechende Programmzeile angezeigt und der Cursor befindet sich auf dem ersten Zeichen des gefundenen Ausdrucks. Ist der Ausdruck im weiteren Listing nicht mehr enthalten, bleibt der Cursor an der Suchstart-Position stehen.

Find Next **<I>**: Suche mit demselben Ausdruck ab Cursorposition fortsetzen.

Exchange **<E>**: Ausdruck ab Cursorposition suchen und gfls. ersetzen. Es erscheint in der Fußzeile eine Aufforderung zur Eingabe eines Suchstrings. Ein gfls. vorher schon eingegebener String wird voreingestellt. Die Eingabezeile kann durch **<Esc>** gelöscht werden. Eine zweite Aufforderung erwartet dann die Eingabe des Strings, durch den der erste ersetzt werden soll. Durch **<Return>** im Anschluß an die String-Eingabe wird dann die Suche ausgelöst (**<Esc>**, dann **<Return>** = Abbruch). Weiteres Suchen und Ersetzen erfolgt dann über **<Strg>+<E>** bzw. **<Ctrl>+<E>**.

Strings innerhalb 'versteckter' Prozeduren werden nicht gefunden oder ersetzt. Wird der Ausdruck gefunden, wird die entsprechende

Programmzeile angezeigt und der Cursor befindet sich hinter letzten Zeichen des veränderten Ausdrucks. Ist der Ausdruck im weiteren Listing nicht mehr enthalten, bleibt der Cursor an der Suchstart-Position stehen.

Exchange **<X>**: Suche und Ersetzen mit denselben Ausdrücken ab Next Cursorposition fortsetzen.

Case **<C>**: *a<>A* Bei der Suche wird Groß-/Kleinschreibung berücksichtigt.

Nocase **<N>**: *a==A* Bei der Suche bleibt Groß-/Kleinschreibung unberücksichtigt.

'Block' - Menü

(Block-Operationen) öffnen: **<F3> oder <Alt>+<B>**

Set Block **<B>**: Blockstart setzen

Set Block **<K>**: Blockende setzen

Copy **<C>**: Block an aktuelle Cursorposition kopieren. Block in Block ist nicht möglich.

Move **<M>**: Block an aktuelle Cursorposition verschieben. Block in Block ist nicht möglich.

Write **<W>**: Block als ASCII-Text speichern (vgl. 'File'-*Write*)

Print **<P>**: Block auf Drucker ausgeben (vgl. 'File'-*Print*)

Hide **<H>**: Block-Start- und Ende-Markierungen aufheben

Delete : Block löschen. Um fatale *'Schnellschüsse'* zu vermeiden, gibt es hierfür keine Tastenfunktion.

'Direct' - Modus an: **<F4> oder <Alt>+<D>**

(s. DER DIREKT-MODUS)

'PgUp'

Seite aufwärts blättern : **<F5>**

'PgDn'

Seite abfwärts blättern : **<F6>**

'Undo'

aktuelle Zeilenänderungen
rückgängig machen : **<F7> oder <Alt>+<U>**

'Insert'/'Overwr'

Zeichen-Eingabe zwischen
'Einfüge'- (Insert) und
'Überschreib'-Modus
(Overwrite) umschalten : **<F8> oder <Alt>+<I>**
bzw. <Alt>+<O>

'View'

letzte Programmseite zeigen: **<F9> oder <Alt>+<V>**

'Run'

aktuelles Programm starten : **<F10> oder <Alt>+<R>**

'Uhrzeit'

am rechten Rand finden Sie ein acht Zeichen langes Feld mit der aktuellen Uhrzeit.

'Pfeile'

Rechts neben der Uhrzeit finden Sie zwei Pfeile. Ein Klick auf den *'Abwärtspfeil'* bewirkt, daß der Cursor *'nach unten'* fährt und gfls. die Seite mitscrollt. Ein Klick auf den *'Aufwärtspfeil'* bewirkt ein Seitenscrolling *'nach unten'*.

Fußzeile ganz links:

In der Fußzeile erscheint jeweils die Nummer der Programmzeile und die Spaltenposition, auf welcher sich der Cursor momentan befindet. Rechts daneben ist der aktuell eingestellte Dateipfad jederzeit sichtbar.

1.4. TASTATUR - FUNKTIONEN

Da es im PC-Bereich zwei Tastaturen gibt (sog. XT- und AT-Tastatur), deren Tasten unterschiedlich beschriftet sind, werden sie hier auch unterschiedliche Tastenbezeichnungen finden. Die AT-Tastatur erkennen Sie dabei an der Umrandung.

Cursor- und zeilenbezogene Funktionen:

Cursor nach rechts	:	**<Rechts-Pfeil**
Cursor nach links	:	**<Links-Pfeil>**
Cursor an den Zeilenanfang	:	**<Pos1>** bzw. **<Home>**
Cursor an das Zeilenende	:	**<Ende>** bzw. **<End>**
Cursor auf Tabulator rechts	:	**<Tab>**
Cursor auf Tabulator links	:	**<Strg>+<Tab>** bzw. **<Ctrl>+<Tab>**
Cursor eine Zeile hoch	:	**<Aufwärts-Pfeil>**
Cursor eine Zeile runter	:	**<Abwärts-Pfeil> oder <Return>**
Cursor beliebig positionieren	:	**Maus-Linksklick** auf gewünschte Position
Cursor eine Seite rückwärts	:	**<Bild-hoch> bzw. <Pg up>** oder **<F5>** oder **<Strg>+<R>** bzw. **<Ctrl>+<R>** oder **Klick** auf *"PgUp"*
Cursor eine Seite vorwärts	:	**<Bild-runter>bzw.<Pgdown>** oder **<F6>** oder **<Strg>+<C>** bzw. **<Ctrl>+<C>** oder **Klick** auf *"PgDn"*
Cursor an Programmanfang	:	**<Strg>+<Pos1>** bzw. **<Ctrl>+<Home>**

Cursor an das Programmende:	**<Strg>+<Ende>** bzw. **<Ctrl>+<End>** oder **<Strg>+<Z>** bzw. **<Ctrl>+<Z>**
aktuelle Änderung löschen (UNDO-Funktion) :	**<Alt>+<U>** oder **<Alt>+<Backspace>** oder **<F7>**
Leerzeile einfügen :	**<Einfg>** bzw. **<Ins>** oder **<Strg>+<N>** bzw. **<Ctrl>+<N>**
Zeichen links löschen :	**<Backspace>**
Zeichen unter Cursor löschen:	**<Entf>** bzw. **<Del>**
Zeilenrest ab Cursorposition : löschen und in Sequenz-Puffer laden	**<Strg>+<P>** bzw. **<Ctrl>+<P>**
Zeilensequenz aus Puffer an : akt. Cursorposition einfügen	**<Strg>+ <O>** bzw. **<Ctrl>+ <O>**
aktuelle Cursorzeile löschen : und in Zeilenpuffer laden	**<Strg>+<Entf>** bzw. **<Ctrl>+<Del>** oder **<Strg>+<Y>** bzw. **<Ctrl>+<Y>**
Pufferzeile (<Strg><Y>) an : akt. Cursorposition einfügen	**<Strg>+ <U>** bzw. **<Ctrl>+ <U>** oder **<Alt>+<Einfg>** bzw. **<Alt>+<Ins>**
Zeilensprung :	**<Strg>+<G>** bzw. **<Ctrl>+<G>**

Weitere Tastatur-Kommandos :

Block-Anfang definieren :	**<Strg>+<B>** bzw. **<Ctrl>+<B>** oder **<Strg>+<K>** dann **<B>** bzw. **<Ctrl>+<K>** dann **<B>**

Block-Ende definieren	:	**<Strg>+<K>** dann **<K>** bzw. **<Ctrl>+<K>** dann **<K>**
letzten Ausdruck ab Cursorposition finden	:	**<Strg>+<F>** bzw. **<Ctrl>+<F>**
letzten Ausdruck ab Cursorposition finden und ersetzen	:	**<Strg>+<E>** bzw. **<Ctrl>+<E>**
Editor-Zeilenmarken setzen	:	**<Strg>+<F1>** bis **<Strg>+<F6>** bzw. **<Ctrl>+<F1>** bis **<Ctrl>+<F6>**

Es können durch **<Strg>** bzw. **<Ctrl>** und die Funktionstasten **F1** bis **F6** bis zu sechs unsichtbare *Marken* im Editor auf beliebige Programmzeilen (jeweils die aktuelle Cursorzeile) gesetzt werden.

Editor-Zeilenmarken anspringen	:	**<Alt>+<F1>** bis **<Alt>+<F6>**

Es kann die durch die jeweilige Funktionstaste markierte Zeile angesprungen werden, falls die hier aufgerufene Markierung vorher durch **<Strg>+<Fx>** bzw. **<Ctrl>+<Fx>** vergeben worden ist.

Cursor an die Position vor dem letzten Programmstart, bzw. vor dem letzten *'Direct'*-Aufruf setzen	:	**<Alt>+<F7>**
Cursor an Error-Position setzen	:	**<Alt>+<F8>**

Wurde das Programm durch einen Error unterbrochen, so wird - falls möglich - eine Zeilenmarke auf die zum Abbruch führende Zeile gesetzt. Diese Zeile kann gfls. durch **<Alt>+<F8>** aufgerufen werden.

Cursor an Position vor dem letzten Such-Start setzen	:	**<Alt>+<F9>**

Cursor an die Position der
letzten Programm-Änderung setzen: **<Alt>+<F10>**

Einzelne PROCEDURE oder
FUNCTION ein- und ausklappen
(*Folding/Unfolding*) :

Cursor auf **PROCEDURE**- bzw. **FUNCTION**-Kopfzeile positionieren, dann **<Alt>+<Q>** (bzw. bei AT's auch **<F11>**) drücken. Die Kopfzeile wird durch **'>'** gekennzeichnet und der Prozedur-Rest verschwindet. Die **'>'**-Kennzeichnung bleibt auch beim Speichern durch *'Save'* oder *'Write'* im Programm-File erhalten und wird beim Laden durch *'Load'* oder *'Merge'* erkannt, sodaß danach wieder nur die Kopfzeile erscheint.

Durch erneutes **<Alt>+<Q>** (bzw. **<F11>**) in der Kopfzeile einer eingeklappten Prozedur - bzw. indem man das **'>'**-Zeichen am Zeilenanfang löscht - wird diese wieder sichtbar.

Alle PROCEDURE's und FUNCTION's
ein- und ausklappen (*Folding/Unfolding*):

Cursor auf **PROCEDURE**- bzw. **FUNCTION**-Kopfzeile positionieren, dann **<Alt>+<W>** (bzw. bei AT's auch **<F12>**) drücken. Die Kopfzeilen aller folgenden Prozeduren und Funktionen werden durch **'>'** gekennzeichnet und die Prozedur-Listings verschwinden. Die **'>'**-Kennzeichnungen bleiben auch beim Speichern durch *'Save'* oder *'Write'* im Programm-File erhalten und werden beim Laden durch *'Load'* oder *'Merge'* erkannt, sodaß danach wieder nur die Kopfzeilen erscheinen.

Durch erneutes **<Alt>+<W>** (bzw. **<F12>**) in der Kopfzeile einer eingeklappten Prozedur wird diese und alle darauffolgenden wieder sichtbar.

'File'-Menü öffnen : **<Alt>+<F>**

'Block'-Menü öffnen : **<Alt>+<S>**

'Search'-Menü öffnen : **<Alt>+<S>**

'Direct'-Modus aufrufen : **<Alt>+<D>**

'Undo'-Funktion aufrufen : **<Alt>+<U>**

'Overwrite/Insert' umschalten : **<Alt>+<O>** bzw. **<Alt>+<I>**

'View' - letzte Programmseite betrachten : **<Alt>+<V>**

'Run' - Programm starten : **<Alt>+<R>**

Erweitertes 'Block'-Menü öffnen : **<Strg>+<K>** bzw. **<Ctrl>+<K>** oder **<Alt>+<B>**

Es erscheint ein ein Block-Menü mit weiteren Optionen. Bis auf zwei Besonderheiten sind die Block-Optionen *'Block Start'* bis *'Block-Move'* mit den unter 'DAS EDITOR-MENÜ' beschriebenen Optionen identisch. Die erste Besonderheit besteht darin, daß hier für die Delete-Option zusätzlich die Tastenfunktion **<Y>** verfügbar ist und die zweite darin, daß das *'Block-Read'* im Gegensatz zum normalen *'Merge'*, Blockmarken um den neu geladenen Programmblock legt.

Die Optionen *'Save GFA'*, *'Quit GFA'* und *'Save & Quit'* sprechen für sich (vgl. 'DAS EDITOR-MENÜ' unter 'File').

Zusätzlich hat man hier noch die Möglichkeit, über die Tasten **<1>** bis **<6>** Zeilenmarken zu setzen.

Optionen-Menü öffnen : **<Strg>+<Q>** bzw. **<Ctrl>+<Q>**

Es erscheint ein Menü mit weiteren Editor- Optionen:

Find : sucht nächsten Ausdruck (wie **<Strg>+<F>**)

Exchange : ersetzt den nächsten gefundenen Ausdruck (wie **<Strg>+<E>**)

Kill to EOL : löscht aktuellen Zeilenrest (*to EndOfLine*) und lädt ihn in den Sequenzpuffer (wie **<Strg>+<P>**)

Output to EOL : fügt den Sequenzpuffer-Inhalt ab aktueller Cursorposition in die Zeile ein (wie **<Strg>+<O>**)

Restore Line	:	restauriert die Zeile und löscht die bisherigen Änderungen (wie **<F7>**)
Goto Block Start	:	setzt den Cursor auf den aktuellen Blockanfang
Goto Block End	:	setzt den Cursor auf das aktuelle Blockende
Goto Mark 0 ... *Goto Mark 9*	:	setzt den Cursor an die jeweilige - falls vorhandene - Zeilenmarke. Die jeweils markierten Zeilennummern sind angegeben. (wie **<Alt>+<Fx>**)

1.5. DER DIREKT-MODUS

Es ist möglich, fast jeden GFA-BASIC-Befehl im Direkt-Modus einzugeben. In diesen Modus gelangen Sie, indem Sie im Editor-Menü das *'Direct'*-Feld mit der Maus anklicken oder die Tastenkombination **<Alt><D>** drücken. Es erscheint dann der beim letzten Programm-Ende oder -Abbruch *'hinterlassene'* Bildschirm und es können in einer direktinterpretierenden Kommando-Eingabezeile die verschiedenen Befehle eingegeben werden. Die Verwendung von Schleifen- und **GOTO**-Anweisungen, **PROCEDURE**-Definitionen, Bedingungsabfragen und ähnlicher Strukturbefehle sind im Direkt-Modus jedoch nicht möglich, da sie sich auf die Programmstruktur beziehen, die ja im Direkt-Modus nicht verfügbar ist.

Die Rückkehr aus dem Direkt-Modus zum Editor erfolgt wahlweise mit **<Return>** innerhalb einer Leerzeile oder der Eingabe des **EDIT**-Befehls, gefolgt von **<Return>**.

GFA-BASIC bietet die Möglichkeit, eigene Editor-Erweiterungen zu schreiben (z.B. Datei-KILL oder Datei-LIST) und diese aus dem Direkt-Modus heraus aufzurufen, da der Befehl **GOSUB** auch hier funktioniert und somit eine Prozedur, die sich im aktuellen Programm befindet - vorausgesetzt, die Syntax ist in Ordnung - aufgerufen werden kann.

Im Direktmodus des GFA BASIC ist ein sogenannter *'History'*-Modus eingebaut, der es erlaubt, sich durch Betätigung der **<Pfeil-hoch>**- bzw. **<Pfeil-runter>**-Cursortaste die letzten acht direkt eingegeben Befehlszeilen anzeigen zu lassen. Die angezeigte Zeile kann dann neu editiert und/oder durch **<Return>** wieder gestartet werden. Außerdem kann durch **<Einfg>** bzw. **<Ins>** zwischen dem *'Einfüge'*- und *'Überschreib'*-Modus gewechselt werden.

1.6. DIE ERSTEN SCHRITTE

Wie bereits einführend angedeutet wurde, richtet sich dieses Buch in erster Linie an BASIC- oder auch bisherige PASCAL-Programmierer, die schon mit den grundlegenden allgemeinen Verfahrensweisen bei der Erstellung von Programmen einigermassen vertraut sind.

Trotzdem möchte ich hier auch Neulingen die Möglichkeit geben, wenigstens die allerersten Schritte zu ihrem ersten - oder auch zweiten - Computerprogramm zu ermöglichen. Die 'gelangweilten' Halb- und Vollprofis mögen mir verzeihen - ich werde mich kurz fassen.

Ich setze hier jedoch schon das Wissen darüber voraus, wie ein Programm von der MSDOS-Kommandoebene oder einer sonstigen Bedieneroberfläche aus gestartet wird. Dieses tun Sie nun mit Ihrem GFA-BASIC. Es erscheint sodann der im Kapitel 'DER EDITOR' beschriebene Bildschirm mit dem Editor-Menü am oberen Rand. Der Text-Cursor (ein kleiner, blinkender Unterstrich, der die aktuelle Texteingabe-Position markiert) steht in der linken, oberen Ecke des noch freien Editorfeldes und Sie können nun sofort mit der Programmeingabe beginnen.

Hier werden schon einige der erfahrenen Programmierer stutzen: *'Was? Mehr muß ich nicht tun?? So einfach ist das??'* - Ja, genau so einfach ist das! Im Gegensatz zu vielen anderen Programmiersprachen und Dialekten müssen Sie sich in GFA-BASIC normalerweise nicht um globale Deklarationen, Includes oder sonstige schwerfällige Verwaltungsakte kümmern, GFA-BASIC erledigt das für Sie. Die allererste Programmzeile, die ein angehendes Programmier-Genie in (fast) allen Programmiersprachen möglichst widerspruchslos zu absolvieren hat, ist:

```
PRINT ''hello world''
```

Nachdem diese Zeile oben links in Ihrem GFA-Editor prangt, klicken Sie im Editor-Menü *'Run'* an oder drücken die Tastenkombination **<Alt><R>**. Der GFA-BASIC-Interpreter schaltet nun auf einen anderen Bildschirm - Ihren Programm-Bildschirm - um und gibt oben links den Satz "hello world" aus. Anschließend erscheint eine kleine Alert-Box, die Sie darauf hinweist, daß das Programm beendet ist. Wenn Sie nun die **<Return>**-Taste drükken oder den *'Return'*-Button der Box mit der Maus anklicken, kehrt GFA-BASIC in den Programm-Editor zurück, wo weitere große Herausforderungen und Abenteuer auf Sie warten. Das ist wie eine Äquatortaufe für Programmierer. Ab diesem Zeitpunkt gehören Sie *'dazu'*. Der erste Schritt ist getan - es kann nur noch aufwärts gehen!!

Eigentlich könnte hier die Einführung enden - wenn da nicht ein kleines Problem wäre: es gibt tatsächlich noch mehr Befehle, die alle dringlichst darauf harren, eingehend untersucht zu werden. Dies kann natürlich nicht in einer Einführung geleistet werden, dazu gibt es ja den großen Befehlsteil hier im Buch.

Wir erweitern hier das obige Prográmmchen um zwei Zeilen. Dazu fahren Sie mit dem Cursor (Sie wissen schon: dem kleinen blinkenden Strich) in die eben geschriebene Programmzeile (**<Aufwärts-Pfeil>** drücken) und drücken dann die **<Einfg>**- bzw. **<Ins>**-Taste. Die PRINT-Zeile rutscht nun um eine Zeile abwärts und darüber entsteht eine Leerzeile. In diese Leerzeile geben Sie nun ein:

```
FOR i%=1 TO 10
```

Um die Zeile abzuschliessen, drücken Sie nun die **<Return>**-Taste. Wenn Sie die Zeile richtig eingegeben haben, springt der Cursor an den Anfang der nächsten Zeile, die dann leer ist, weil die PRINT-Zeile erneut um eine Zeile abwärts rutscht. Sie könnten nun in diese Leerzeile weiteren Programmtext eingeben. Das möchten wir aber in diesem Fall nicht, sondern die PRINT-Zeile soll die zweite Zeile bleiben. Daß sie nun in Zeile drei steht und Zeile zwei leer ist, soll uns hier nicht weiter kümmern. Das ist ein kosmetisches Problem. Den Interpreter kümmert's auch nicht, er ignoriert leere Zeilen einfach. Falls Sie es doch für notwendig halten, die leere Zeile zu entfernen, drücken Sie nun - sofern der Cursor noch in der leeren zweiten Zeile steht - die Kombination **<Strg><Entf>** bzw. **<Ctrl><Del>**. Die PRINT-Zeile rutscht nun eine Zeile aufwärts und die Leerzeile verschwindet. Damit wäre die optische Ordnung wieder hergestellt.

Nun steht der Cursor immer noch in der zweiten Zeile. Drücken Sie nun einmal die **<Abwärts-Pfeil>**-Taste, um ihn in der dritten Zeile zu positionieren. Ist er in der dritten Zeile? Gut, geben Sie nun folgende Zeile ein:

```
NEXT i%
```

Ihr Programm müßte nun so aussehen:

```
FOR i%=1 TO 10
  PRINT ''hello world"
NEXT i%
```

Wenn Sie nun *'Run'* anklicken oder **<Alt><R>** drücken, wird das Programm nun zehn mal - oben in der ersten Zeile beginnend - "hello world" untereinander schreiben. Die dann folgenden *'Programmende'*-Box kennen Sie ja schon.

Damit haben Sie bereits eine sogenannte 'Schleifenkonstruktion' kennengelernt. Die beiden Zeilen `'FOR i%=1 To 10'` und `'NEXT i%'` bilden eine Einheit und bewirken, daß alle Programmzeilen,

die zwischen diesen beiden Anweisungen stehen, so oft ausgeführt werden, wie in der Eingangsanweisung 'FOR...' angegeben wurde - hier also '1 TO 10' = zehn Mal.

Würden Sie versuchen, eine FOR-Anweisung ohne die dazugehörige NEXT-Anweisung (das Schleifenende bzw. der Wendepunkt) zu starten, so würde GFA-BASIC Sie auf dieses Versäumnis hinweisen.

Eine Schleifenkonstruktion kann auch so aussehen:

```
DO
  PRINT ''hello world''
LOOP
```

Die vorherige FOR-Schleife wird als eine 'Zählschleife' bezeichnet, da sie implizit einen Zählvorgang durchführt und die Schleife beendet, sobald der angegebene Endwert erreicht ist. Die DO...LOOP-Schleife dagegen ist eine sogenannte 'Endlos-Schleife', die eigentlich nie verlassen wird. Es sei denn, daß explizit eine Abbruchbedingung gestellt wird. Das kann folgendermaßen erreicht werden:

```
DO
  PRINT ''hello world''
  a$=INKEY$
  EXIT IF LEN(a$)
LOOP
```

Das Programm schreibt nun solange "hello world", bis eine beliebige Taste gedrückt wird. Der Befehl INKEY$ stellt fest, ob eine Taste gedrückt wird, ohne daß er dabei das Programm unterbricht. Wurde eine Taste gedrückt, wird in der 'String'-Variablen 'a$' ein Text-String geliefert, der dem Code der gedrückten Taste entspricht. Wird keine Taste gedrückt, wird von INKEY$ einfach *'Nichts'* zurückgegeben. Die Funktion LEN() überprüft in der EXIT IF-Zeile, ob der von INKEY$ gelieferte String in *'a$'* überhaupt eine Länge aufweist. Ist das der Fall, so wurde eine Taste gedrückt und die EXIT IF-Bedingungsabfrage wird gültig, sodaß das Programm hinter der LOOP-Zeile fortgesetzt wird. Da sich in diesem Fall dort kein weiterer Programmtext befindet, wird das Programm beendet. Kann man auf den Code der gfls. gedrückten Taste verzichten, so läßt sich die Konstruktion auch verkürzen, indem die gesamte Abfrage nur in der EXIT IF-Zeile stattfindet:

```
EXIT IF LEN(INKEY$)
```

Es gibt noch einige Schleifenkonstruktionen mehr, wobei es auch teilweise möglich ist, diese miteinander zu kombinieren.

Anschließend will ich Ihnen noch zeigen, was eine sogenannte ***'Prozedur'*** (gfls. 'Routine', 'Unterprogramm' oder unter Umständen

auch 'Modul' genannt) ist. Eine Prozedur ist eine abgeschlossene *'Struktur'* innerhalb des Gesamtprogramms oder einer anderen *'Routine'*. Ein solches *'Unterprogramm'* erledigt im allgemeinen Aufgaben, die immer wiederkehren und die - anstatt sie jedesmal neu zu schreiben - in einer Prozedur einmalig definiert werden und dort nötigenfalls durch einen Prozeduraufruf von einer beliebigen Programmstelle aus aufgerufen, also zu ihrer Tätigkeit veranlasst werden:

```
SCREEN 18                   // VGA-Grafik, geht auch in
REPEAT                      //        CGA oder HGC
  xp&=RAND(_X-100)+50       // horiz. Parameter berechnen
  yp&=RAND(_Y-100)+50       // vert. Parameter berechnen
  lines(xp&,yp&,RAND(50)+30,RAND(50)+30)
  '                         // Aufruf m. Parametern
UNTIL MOUSEK or LEN(INKEY$)
'                           // Abbruch bei belieb. Taste
'---------------------------------
PROCEDURE lines(x%,y%,rx%,ry%)
  '                         // Prozedurkopf mit Parameter
  FOR i&=0 TO 359 STEP 20//Schleifen-Start (in Grad)
    LINE x%+COSQ(i&)*rx%,y%,x%,y%+SINQ(i&)*ry%
    '                       // Linie zeichnen
  NEXT i&                   //  Schleifen-Wendepunkt
RETURN                      //  Rücksprung z. Hauptprogr.
```

Sollten Sie zu den einzelnen Befehlen und Funktionen (in Großschrift), sowie den Variablen (in Kleinschrift) Fragen haben, schauen Sie bitte unter der entsprechenden Befehls-/ Funktionsbeschreibung, sowie im Kapitel 'VARIABLEN-TYPEN' nach. Diese kleine Eigenleistung bewirkt einen größeren Lerneffekt, als wenn ich nun dieses Programm hier in seinen Feinheiten beschreiben würde (wie sagt man so schön: *'nur selber essen macht satt'*).

Diese kleine Prozedur prozudiert ...äh...produziert - wie sollte es bei dieser Gelegenheit auch anders sein - ein grafisches Spiegelei ...*???*...äh... ´tschuldigung, eine grafische Spielerei meine ich natürlich. Solche Demo-Routinen sind deshalb sehr begehrt, weil sie extrem kurz gehalten werden können und auch kein tieferer Sinn dahinter stecken muß. Weitschweifende Erklärungen über Zweck und Nutzen erübrigen sich auch. Leider scheint es in der Computer-Literatur - bis auf einige positive Ausnahmen - üblich zu werden, weitgehend sinnlose Demo-Routinen dieser Art zu *'verwursten'*. Schauen Sie sich bitte unser in 5/92 erscheinendes COLID-*'PREMIUM-Buch zum GFA BASIC für MSDOS'* an. Dieses Buch steckt prallvoll mit nützlichen und allgemein definierten Hilfs-, Arbeits- und Trickroutinen (zwei inliegende Disketten), die Ihnen die Programmierarbeit ganz erheblich erleichtern können.

Sie haben im Laufe dieser Einführung nun schon einige Variablen-Zuweisungen kennengelernt:

```
a$=INKEY$
```

oder:

```
xp&=RAND(_X-100)+50
```

Die Zuweisung von Werten, Ausdrücken, Texten oder sonstigen wichtigen Daten zu den verschiedenen Variablentypen nimmt einen erheblichen Raum in jedem Computerprogramm ein. Man kann sogar sagen, je mehr das Programm allgemein verfügbare Variablen enthält, umso mehr und leichter kann das Programm gesteuert werden. Jede *'globale'* Variable ist eine *'Schnittstelle'* zum Programm, d.h., daß es hierüber möglich ist, 'Informations-Leitungen' aus dem Programm herauszuführen und dem Anwender zur Verfügung zu stellen.

Das weiter oben aufgeführte Demo-Programm 'lines' soll hier als Beispiel dienen:

Der Prozedur-Kopf 'PROCEDURE lines' enthält die Aufnahme-Variablen für die horizontale Mitte (x%), die vertikale Mitte (y%), sowie für den horizontalen (rx%) und vertikalen (yr%) Radius der Grafik. Wie unschwer zu erkennen ist, wird als Gradeinteilung der volle Kreis (FOR i&=0 TO 359) und als Schrittweite der Wert 20 (STEP 20) verwendet. Ändert man an dieser Routine nichts, so wird die Grafik immer im vollen Umkreis mit 20-Grad-Schritten gezeichnet.

Durch eine einfache Umstellung des Programms ist es nun möglich, dem Aufrufer der Routine (in diesem Fall ja Sie selbst) drei weitere Einstellungsmöglichkeiten anzubieten. Wenn nämlich stat der drei fest vorgegebenen Werte '0', '359' und '20' ebenfalls Variablen eingesetzt würden, so könnte man diese in der Kopfzeile der Prozedur zusätzlich als Parameter-Aufnahmevariable aufführen:

```
PROCEDURE lines(x%,y%,rx%,ry%,beg&,end&,schritt&)
  FOR i&=beg& TO end& STEP schritt&
    LINE x%+COSQ(i&)*rx%,y%,x%,y%+SINQ(i&)*ry%
  NEXT i&
RETURN
```

Rufen Sie nun diese Prozedur z.B. folgendermaßen auf:

```
lines(xp&,yp&,RAND(50)+30,RAND(50)+30,90,179,3)
```

so werden Sie eine Änderung in der Ausführung feststellen.

Bedenken Sie generell diese global gedachte Definition von Daten

innerhalb Ihrer eigenen Programme so konsequent wie möglich. Nach einiger Erfahrung werden Sie sich wundern, wie komplex, vielfältig und professionell Sie Ihre - anfänglich auch noch so kleinen - Programme ausarbeiten können, ohne das Rad jedesmal neu erfinden zu müssen. Außerdem haben Sie - sofern Sie diese Technik anwenden und perfektionieren wollen - in der Endkonsequenz schon einen großen Schritt in Richtung *'OOP'* (objektorientierte Programmierung) getan. Wer in der heutigen Zeit moderne Programme entwickeln und nicht als EDV-Dinosaurier abgetan werden will, der wird nicht um diese spezielle Art der Programmtechnik herumkommen. Wenn Sie *'normal geradeaus'* programmieren möchten, so vergessen Sie das eben gesagte. Hauptsache, Ihnen bringt das Programmieren Spaß und den gewünschten Erfolg.

Übrigens:

Die horizontale Ausdehnung auf einem Bildschirm wird in der Computerei auch die 'X'-Dimension oder 'X'-Richtung genannt. Sie verläuft waagerecht von links nach rechts: von Null bis zur Bildschirmbreite in Pixel. Der Begriff 'pixel' stammt aus dem englischen und bedeutet soviel wie 'Bildelement' (picture-element). Die vertikale Ausdehnung (in GFA BASIC unter MSDOS von oben nach unten verlaufend: von Null bis zur Bildschirmhöhe in Pixel) ist dagegen die 'Y'-Dimension oder auch 'Y'-Richtung. Es gibt allerdings auch Flächenorientierungssysteme, die den Bildschirm von unten nach oben einteilen, worum Sie sich in GFA BASIC unter MSDOS allerdings nicht weiter zu kümmern brauchen.

So, das waren die allerersten Schritte, bei denen ich in der gegebenen Kürze versucht habe, Ihnen das Händchen zu halten. Weitere Erfolgserlebnisse stehen noch vor Ihnen, wenn Sie nun versuchen, den Sinn und die Arbeitsweise der verschiedensten Befehle des GFA-BASIC zu ergründen. Dabei werden auch Sie nicht vor einem unangenehmen Vorgang gefeit sein, den man erfrischend und einfach *'Absturz'* nennt. Damit ist ein Zustand gemeint, bei dem der Computer aus vielerlei Gründen (oder auch nicht, keiner weiß das immer so ganz genau) komplett seinen Dienst quittiert und Sie damit nachhaltig zum Drücken der Reset-Taste animiert. Computer sind auch nur Menschen. Lassen Sie sich davon nicht beirren - nur dem Geduldigen und Fleißigen blüht der Erfolg. Sichern Sie immer wieder rechtzeitig Ihre Programme auf Diskette oder Festplatte, das erspart Ihnen bei einem Absturz viele, viele nervenaufreibende Reparaturarbeiten am Programm.

1.7. VARIABLEN - TYPEN

variable=8Byte-IEEE-Realwert

Variablennamen ohne Kennung bzw. mit der Kennung '#' (z.B.: Var#) werden **'Fließkomma'**-Variablen (auch **'Real'**-Variablen) genannt. DieserVariablentyp benötigt zu seiner Speicherung im IEEE-Double-Format 8 Byte Speicherplatz. Im Dezimalbereich (Nachkommastellen) kann so eine Genauigkeit von bis zu maximal 13 Stellen eingehalten werden. Nimmt der ganzzahlige Anteil mehr als 13 Stellen ein, wird der Variablenwert in das Exponentialformat konvertiert. In diesem Format können dann Werte mit einem ganzzahligen Anteil von bis zu 308 Stellen erfaßt werden.Als größter darstellbarerWert gilt 1.67E+308 und als kleinster Wert 2.2E-308.

Wurde durch DEFxxx (**DEFBIT, DEFBYT** etc.) kein anderesVariablenformat eingestellt,so wird jederVariablenname, der innerhalb des Programms ohne Kennung eingegeben wird, als eine Realvariable angesehen.

variable&=2Byte-Integerwert

Variablennamen mit dem Postfix '&' (z.B.:Var&) gelten als 2 Byte-Integer-Variablen (**'Word-Integer'**). Jeder diesem Variablentyp zugeordneteWert wird auf seinen ganzzahligen Anteil reduziert.D.h., evtl.auftretenden Nachkommastellen werden 'integriert'. Zu seiner Speicherung benötigt dieser Typ 2 Byte Speicherplatz. Es können Werte im Bereich von -32768 (= -2^15) bis zu +32767 (= 2^15-1) verarbeitet werden. Werte außerhalb dieses Bereichs werden mit einer 'Überlauf'-Meldung moniert.

variable%=4Byte-Integerwert

Variablennamen mit dem Postfix '%' (z.B.:Var%) werden als 4-Byte-Integer-Variablen (**'Long-Integer'**) interpretiert.Wie bei der Word-Integer-Variablen werden auch hier die zugeordneten Werte auf ihren ganzzahligen Anteil reduziert. Evtl. auftretende Nachkommastellen werden also ebenfalls 'integriert'. Zu seiner Speicherung benötigt dieser Typ 4 Byte Speicherplatz. Es können Werte im Bereich von -2147483648 (= -2^31) bis +2147483647 (= 2^31-1) verarbeitet werden. Werte außerhalb dieses Bereichs werden mit einer 'Überlauf'-Meldung moniert.

variable|= I Byte-Integerwert

Variablennamen mit der Kennung '|' (z.B.Var|) gelten als vorzeichenlose 1-Byte-Integer-Variablen. Auch bei diesem Typ werden zugeordnete Werte ihres Nachkomma-Anteils entledigt. Zu seiner Speicherung wird für diesen Variablentyp 1 Byte Speicherplatz benötigt. Es können nur positive Ganzzahlwerte im Bereich von 0 bis 255 hiermit verarbeitet werden. Werte außerhalb dieses Bereichs werden auch hier als Überlauf angesehen.

variable!=Wahrheitswert TRUE (-1) oder FALSE (0)

Variablennamen mit der Kennung '!' (z.B.Var!) sind sogenannte **'Bool'** - Variablen. Dieser Variablentyp kann ausschließlich die Werte 0 (= FALSE) oder -1 (<> 0 =TRUE) annehmen. Es werden zu seiner Speicherung als Einzelvariable 2 Byte benötigt. Im Gegensatz zu anderen Typen hat diese Variable innerhalb von Feldern jedoch einen anderen Speicherbedarf als den einer Einzelvariablen. Als Feld-Variable benötigt sie nur 1 Bit (!) je Element.

variable$="Text"

Variablennamen mit der Kennung '$' (z.B.Var$) werden als Text-Variablen (sog. **'String'**-Variablen) interpretiert. Eine Variable dieses Typs kann bis zu 32767 Textzeichen (1 Byte je Zeichen) aufnehmen.

Für diesen Variablentyp wird innerhalb einer internen Zeiger-Tabelle jeweils ein String-Zeiger (**'Pointer'**) von 4 Byte eingerichtet, dessen Speicheradresse durch die Funktion ARRPTR() ermittelt werden kann:

```
zeiger_adresse%=ARRPTR(string$)
```

In dieser Tabelle befindet sich ein Longword-Zeiger auf den jeweiligen String-**'Descriptor'** (String-'Beschreiber'). Dieser Descriptor beginnt wiederum immer als sogenannter **'Header'** (Kopf) 6 Bytes vor dem ersten Zeichen des dazugehörigen Textstrings:

```
header_adresse%=LONG{ARRPTR(string$)}
```

Die ersten 4 Bytes dieses Headers zeigen wiederum auf den dazugehörigen Platz innerhalb der beschriebenen internen Zeigertabelle:

```
zeiger_adresse%=LONG{VARPTR(string$)-6}
```




Die Bytes 5 und 6 des Descriptors enthalten im CARD-Format (2Byte) die Länge des eigentlichen Textes.

```
string_laenge%=CARD{VARPTR(string$)-2}
```

oder

```
string_laenge%=CARD{LONG{ARRPTR(string$)}+4}
```

Die Adresse des ersten Zeichens des eigentlichen Textes kann entweder durch die Funktion VARPTR()

```
string_adresse%=VARPTR(string$)
```

oder durch Auslesen des Tabellenzeigers

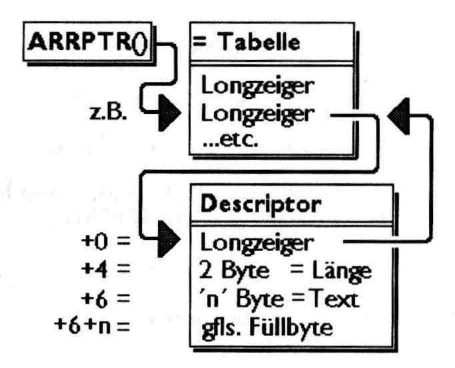
```
string_adresse%=LONG{ARRPTR(string$)+6}
```

ermittelt werden.

Hat der String eine ungerade Länge, wird hinter das String-Ende ein - für das Programm unsichtbares - Füllbyte gesetzt, das gewährleisten soll, daß der Speicherbereich fürdie nächste Textvariable (bzw. deren Header) an einer geraden Adresse beginnt.

Benötigter Speicherplatz:

```
    4 Byte Tabellenzeiger
+   6 Byte für Descriptor-Header
+   1 Füllbyte (gfls.)
+   'm' Bytes ('m' = Stringlänge)
+   'n' Bytes ('n' = Zeichenanzahl des Namens)
-------------------------------------------------
=   11+m+n Bytes
=================================================
```



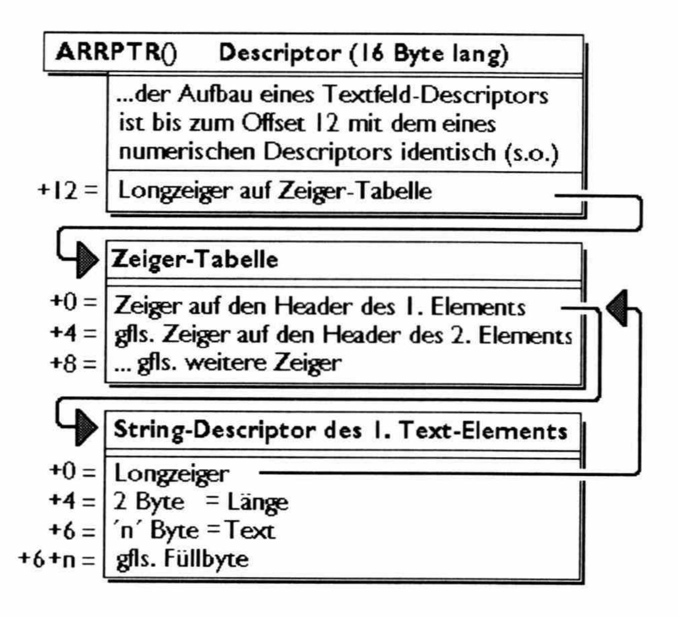
Vektoren und Matrizen

Mit jedem der angeführten Variablentypen lassen sich auch eindimensionale **'Vektoren'** (auch: Arrays) oder mehrdimensionale **'Felder'** (auch: Matrizen) bilden ein Longword, das dann auf die Startadresse des ersten Elementes des Feldes zeigt.

	ARRPTR() = Descriptor (16 Byte lang)
+0 =	2 Byte = Anzahl der Dimensionen
+2 =	2 Byte = Elementeanzahl für Dim. 1
+4 =	2 Byte = gfls. Elementeanz. f. Dim. 2
+6 =	2 Byte = gfls. Elementeanz. f. Dim. 3
+8 =	2 Byte = gfls. Elementeanz. f. Dim. 4
+10 =	2 Byte = gfls. Elementeanz. f. Dim. 5
+12 =	Longzeiger auf das 1. Element
+16 =	gfls. nächster Feld-Descriptor
	... etc. wie oben

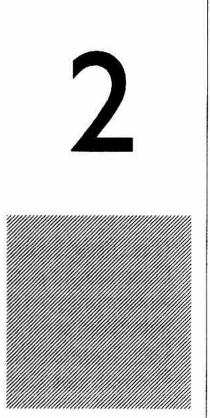
In Bezug auf den Longzeiger bei Offset 12 unterscheiden sich die Descriptoren für Stringfelder von denen für numerische Felder. Bei Stringfeldern weist dieser Zeiger im Descriptor auf eine Tabelle, welche der Reihe nach in

jeweils 4 Byte die Zeiger auf die oben schon beschriebenen String-Header der einzelnen Elemente beinhaltet.



Die Adressen aller Variablen (auch von Feldvariablen-Elementen) lassen sich zusätzlich auch durch die Funktion **VARPTR()** ermitteln (z.B.: `PRINT VARPTR(A%(4,1))`).

Ein- und Ausgabe-Befehle



2. EIN- UND AUSGABE - BEFEHLE

2.1. DATEN - EINGABE

FORM INPUT — Formatierte Stringeingabe

```
FORM INPUT Anzahl,Var$
```

Ermöglicht eine Stringeingabe mit vorgegebener Zeichen-*'**Anzahl**'*. Die Eingabe wird bei Erreichen der max. Zeichenanzahl nicht abgebrochen, sondern es ertönt ein Signalton und die Eingabe muß mit <Return> quittiert werden. Anschließend wird sie der Variablen ***'Var$'*** zugewiesen. Maximale Eingabelänge = 255 Zeichen.

Die Eingabe kann mit <Del>, <Backspace> (bzw. <-) und den vier Pfeil-Tasten editiert werden. Die Sonderzeichen-Eingabe erfolgt wie bei INPUT.

FORM INPUT AS — Formatierte Stringeingabe m. Vorgabe

```
FORM INPUT Anzahl AS Var$
```

Gibt den Inhalt von ***'Var$'*** an der aktuellen Cursor-Position aus und ermöglicht dessen erneute Edition.

Der Parameter ***'Anzahl'*** gibt die Zahl der Zeichen von ***'Var$'*** (vom Anfang ausgehend) an, die ausgegeben werden sollen. Nach <Return> wird ***'Var$'*** komplett durch die Neueingabe ersetzt.

INPUT { INP } — Dateneingabe

```
INPUT [''Text'';,] Var [,Var2,...]
INPUT #Kanal,Var [,Var2,...]
```

Das Programm wird unterbrochen und der Anwender zur Dateneingabe aufgefordert. ***'Text'*** ist eine beliebiger String (in Anführungsstrichen), der optional einen Eingabekommentar enthält. ***'Var'*** wird nach Eingabe-Abschluß die Eingabe zugewiesen. Das Trennzeichen zwischen ***'Text'*** und ***'Var'*** be- stimmt die Position des Cursors im Anschluß an ***'Text'***:



; Fragezeichen und Leerzeichen hinter dem Text

, Eingabe beginnt direkt rechts neben dem Text

Die Eingabe muß mit <Return> bestätigt werden. Passen die eingegebenen Zeichen nicht zum Datentyp der angegebenen Variablen, ertönt ein akustisches Signal und die Eingabe muß wiederholt werden.

Ein Komma in der Dateneingabe bewirkt nur Berücksichtigung des Teils vor dem Komma, bzw. eine Datentrennung, falls auf den INPUT-Befehl mehrere, durch Kommata getrennte Variablen folgen. In diesem Falle können Daten auch durch <Return> getrennt werden. Sollen auch Kommas in die Eingabe mit einbezogen werden, sind diese in Anführungsstriche zu setzen und gesondert einer Variablen zuzuordnen.

Bei Stringeingaben ist die Länge auf 255 Zeichen beschränkt. Die Eingabe kann mit <Del>, <Backspace> und den vier Pfeil-Tasten korrigiert werden.

Die Eingabe von Sonderzeichen über Tastatur kann folgendermaßen vorgenommen werden:

- <Alternate>-Taste gedrückt halten und über den Ziffernblock den ASCII-Wert des Zeichens eingeben. Nach Loslassen der <Alt>-Taste wird das Zeichen in die Eingabe übernommen.

- <Control> (bzw. <Strg>) und <S> Taste gleichzeitig drücken und wieder loslassen. Danach beliebige Taste drücken.

- <Control> (bzw. <Strg>) und <A> Taste gleichzeitig drücken und wieder loslassen. Danach beliebigen ASCII-Code eingeben. Zeichen mit einem ASCII-Wert zwischen 0 und 32 müssen nach Eingabe des Wertes mit <Return> quittiert werden.

Die zweite Syntaxform gibt in ***'Kanal'*** die Datei an, aus welcher die Daten gelesen werden sollen. Die Variablen sind vom Typ her entsprechend den zu lesenden Daten anzugeben.

INPUT$ — Zeichenketteneingabe

```
Var$=INPUT$(Anzahl [,#Kanal])
```

Das Programm wird unterbrochen und die unsichtbare Eingabe erwartet. Mit ***'Anzahl'*** wird eine maximale Eingabelänge angegeben. Ein Quittieren mit <Return> ist nur notwendig, wenn die eingegebene Stringlänge die Längenvorgabe ***'Anzahl'*** nicht erreicht. Die Edition der Eingabe kann - unsichtbar - erfolgen, wie unter INPUT beschrieben. Der optionale Zusatz ***'#Kanal'*** bewirkt, daß die angegebene Zeichenanzahl aus der Datei mit der Kanalnummer ***'Kanal'*** gelesen wird. Der String kann maximal 32767 Zeichen aufnehmen.

LINE INPUT — Zeichenketteneingabe

```
LINE INPUT [''Text'';,] Var$ [Var2$,...]
LINE INPUT #Kanal, Var$ [Var2$,...]
```

Kommata werden in den einzugebenden Text mit aufgenommen. Bei Mehrfach-Eingaben können die einzelnen Strings nur durch Betätigung der <Return>- Taste voneinander getrennt werden. Die Strings werden der (den) jeweiligen Variablen ***'Var$'*** zugeordnet. Der optionale Zusatz ***'Text'*** kann einen beliebigen Eingabe-Kommentar enthalten. Editionsmöglichkeiten wie bei INPUT. Die zweite Syntaxform liest die Daten aus der in ***'Kanal'*** angegebenen Datei.

2.2. DATEN - AUSGABE

PRINT { ? oder P } — Daten ausgeben

```
PRINT [AT(S,Z)] [,;'] [''Text''] [,;'] [Var] [AT(S,Z)]...
                                    ... [,;'] [Expr] [,;']
```

PRINT ohne Angabe von Daten, Texten oder Formatzeichen bewirkt eine Leerzeilenausgabe, die mit einem *'Carriage-Return'* (**CR: CHR$(13)**) und einem *'Line Feed'* (**LF: CHR$(10)**) abgeschlossen wird.

Formatierungszeichen zur Modifikation der Ausgabe:

; 'CR' und 'LF' werden unterdrückt, die folgende Ausgabe schließt sich an das letzte Zeichen der vorhergehenden PRINT-Ausgabe an. Ist die ausgegebene Zeile länger als 80 Zeichen und **WRAP ON** ist aktiv, so tritt ein Zeilenüberlauf ein.

, Die nächste Bildschirm-Ausgabe beginnt mit ihrem ersten Zeichen an der nächsten von fünf horizon talen Tabulatorpositionen (1,17,33,49,65).

' An Stelle dieses Apostrophs wird ein Leerzeichen ausge geben. Sonst hat dieses Zeichen die gleiche Funktion wie ';'.

Durch ***AT(S,Z)*** kann der Cursor an einer bestimmten Bildschirmposition (Spalte,Zeile) plaziert werden. Der Startindex ist in beiden Dimensionen 1 (PRINT AT(1,1)).AT(0,0) ist identisch mit AT(80,25).

Beispiele:

```
PRINT x
PRINT ''Text''
PRINT AT(2,2);
PRINT AT(CRSCOL,CRSLIN-4);
PRINT AT(10,6);''Text''''''''Text'',Var;Expr;
PRINT Expr1,Expr2,Expr3
PRINT (12*6+SQR(55.7)/2.31)<<((2^17) MOD 127-x)
```

Beachten Sie bitte auch das Beispiel zu **XLATE$()**.

PRINT USING { P USING } Daten formatiert ausgeben

```
PRINT [AT(S,Z)] USING ''Format'' [,;'] [Var...]
```

In ***'Format'*** ist ein Formatstring zu übergeben, der die gewünschten Formatangaben enthält. Entsprechend dieser Angaben werden die Texte und Werte zur Ausgabe vorbereitet und ausgegeben. Statt eines Textausdruckes kann auch eine Stringvariable übergeben werden, die die Formatangaben enthält.

Folgende Formatsymbole sind möglich:

= Platzhalter für eine Ziffer
`z.B. PRINT USING ''######'',Int(31421/3)`

. = Position des Dezimalpunktes
`z.B. PRINT USING ''######.####'',31421/4`

+ = Ausgabe auch des positiven Vorzeichens (nur an erster Position des Formatstrings).
`z.B. PRINT USING ''+######.####'',31421/4`

- = Platzhalter für negatives Vorzeichen (nur an erster Position des Formatstrings).
`z.B. PRINT USING ''-#####.######'',31421/-4`

* = Füllzeichen für alle angegebenen Vorkommastellen, die nicht von dem auszugebenden Wert belegt wer den. Sonst wie #.
`z.B. PRINT USING ''*######.####'',31421/1.4`

Hinter dem Dezimalpunkt verwendet, werden soviele * ausgegeben, wie angegeben sind und die aus zugebende Zahl wird real auf die gewünschte Stelle gerundet.
`z.B. PRINT USING ''*######.#****'',31421/1.4`

$** = Voranstellung eines **$ (nur direkt vor dem ersten #).
`z.B. PRINT USING ''$######.##'',31421/1.4`

, = Einfügen eines Kommas (Tausendertrennung)
`z.B. PRINT USING ''##,###,###.###'',3422*2711.7`

^ = Ausgabe im Exponentialformat (nur ein Vorkomma-'#' möglich). Führende # stehen hier für die Stellen des Basis- Anteils und ^ für die Exponentenstellen (E+xxxx). Überflüssige Basis-Stellen werden mit 0 gefüllt.
`z.B. PRINT USING ''#.########^^^^'',13711*64`

! = Das erste Zeichen eines Strings wird ausgegeben
`z.B. PRINT USING ''!sing !n !FA'',''Uhr'',''ijkl'',''Grün''`

& = Gesamtstring wird ausgegeben
`z.B. PRINT USING ''&-BASIC'',''GFA''`

\..\ = Ausgabe von sovielen Zeichen des Strings, wie Länge von \..\ (incl. Backslashs)
`z.B. PRINT USING ''\...\ BASIC ?'',''Alles OK?''`

_ = (Tiefstrich) Interpretiert das hierauf folgende Using-Formatzeichen nicht als solches, sondern gibt es als ASCII-Zeichen aus.
`z.B. PRINT USING ''Abc _### _\ &'',44,''XYZ''`

Zur ***AT(S,Z)***-Option s. unter **PRINT AT(S,Z)**.

Die Zahlendarstellung kann durch **MODE** variiert werden.

PRINT ATXY { P ATXY } Daten positioniert ausgeben

```
PRINT ATXY(S,Z) [,;'] [''Text''] [,;'] [Var] [AT(S,Z)]...
                                 ... [,;'] [Expr] [,;']
```

Die Funktion dieses Befehls ist mit **PRINT AT(S,Z)** identisch. Weitere Beschreibung s. dort.

PRINT ATYX { P ATYX } Daten positioniert ausgeben

```
PRINT ATYX(S,Z) [,;'] [''Text''] [,;'] [Var] [AT(S,Z)]...
                                 ... [,;'] [Expr] [,;']
```

Die Funktion dieses Befehls ist mit **PRINT AT(S,Z)** bzw. mit **PRINT ATXY(S,Z)** identisch, nur daß 'S,Z' zu 'Z,S' bzw. XY zu YX vertauscht wird. Weitere Beschreibung s. dort.

SPC() Leerzeichen ausgeben

```
SPC(Anzahl)
```



'Anzahl' gibt die Anzahl auszugebender Leerzeichen an. SPC() ist nur im Zusammenhang mit **PRINT** einsetzbar (z.B. `PRINT ''Text<'';SPC(12);''>Text''`).

WRITE { WR } Daten ausgeben

```
WRITE [#Kanal,] [''Text'' [Var] [,;'] [Expr] [,;']...
                              ... [,Var,Expr[,;']]
```

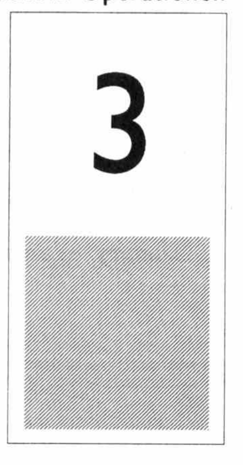
WRITE hat eine mit **PRINT** vergleichbare Funktion. Die Anführungszeichen eines übergebenen Strings bzw. von Ausdrücken oder Variableninhalten, sowie Kommata, welche die einzelnen Listenausdrücke voneinander trennen, werden allerdings - anders als bei **PRINT** - durch WRITE ebenfalls ausgegeben. Bei Verwendung der Option ***'Kanal'*** werden die Daten in die geöffnete Datei mit dem angegebenen Index geschrieben.

Der Sinn dieser Option ist das formatgerechte, auf den **INPUT#**-Befehl zugeschnittene Speichern von Werten und Texten. Sollen nämlich mit dem **INPUT#**-Befehl mehrere Werte oder Strings gleichzeitig eingelesen werden, so müssen die Einzeldaten durch

Kommata voneinander getrennt sein. Da WRITE# Kommata an den entsprechenden Platz in der Datei schreibt, können die Daten beim Einlesen mit **INPUT#** unterschieden und den dabei angegebenen Variablen zugeordnet werden.

Notizen:

Text Bildschirm Operationen



3. TEXTBILDSCHIRM - OPERATIONEN

3.1. CURSOR - NACHFRAGE

CRSCOL aktuelle Cursorspalte liefern

```
Var=CRSCOL
```


Es wird der aktuelle Spalten-Index (X-Position) des Textcursors geliefert.

CRSLIN aktuelle Cursorzeile liefern

```
Var=CRSLIN
```


Es wird der aktuelle Zeilen-Index (Y-Position) des Textcursors geliefert.

POS() CR-bezogene Zeichenspalte ermitteln

```
Var=POS(Dummy)
```


Liefert einen Wert von 0 bis 255, der die aktuelle Stringposition des Cursors seit der letzten Ausgabe eines *CR's* (Wagen-Rücklauf) angibt. Diese Position muß nicht immer mit der realen Cursor-Position identisch sein, da die bildschirm-bezogenen Cursor-Position einen Wert von max. 80 annehmen kann. Für die Berechnung von POS() werden ausschließlich Textzeichen beachtet.

Das Ergebnis kann durch Steuerzeichen beeinflußt sein. Ein im ausgegebenen String enthaltenes *'Carriage Return'* (**CHR$(13)**) setzt den POS()-Zähler auf Null und ein *'Backspace'*-Zeichen (**CHR$(8)**) vermindert POS() um 1.

'Dummy' ist ein beliebiger Blind-Wert ohne weitere Bedeutung.

3.2. CURSOR - POSITIONIERUNG

HTAB { HT } — aktuelle Cursorspalte bestimmen

```
HTAB Spalte
```

Positioniert den Text-Cursor in der angegebenen ***'Spalte'***.

VTAB { VT } — aktuelle Cursorzeile bestimmen

```
VTAB Zeile
```

Positioniert den Text-Cursor in der angegebenen ***'Zeile'***.

LOCATE { LOCAT } — Cursor positionieren

```
LOCATE S,Z
```

Positioniert den Text-Cursor in Spalte ***'S'*** und Zeile ***'Z'***.

LOCAXY — Cursor positionieren

```
LOCAXY S,Z
```

Positioniert den Text-Cursor in Spalte ***'S'*** und Zeile ***'Z'***.

LOCAYX — Cursor positionieren

```
LOCAYX Z,S
```

Positioniert den Text-Cursor in Zeile ***'Z'*** und Spalte ***'S'***.

TAB() — Tabulator setzen

```
TAB(Position)
```



Bestimmung einer horizontalen Tabulatorposition, an welcher die darauffolgende **PRINT**- oder **WRITE**-Anweisung ausgeführt wird.

'Position' liegt im Bereich von 1 bis 255. Befindet sich der Cursor hinter der zuletzt angegebenen Tabulatorposition und liegt der Wert einer sich anschließenden TAB-Anweisung zwischen 256 und der aktuellen Cursorposition, wird dieser Tabulator in derselben Zeile ausgeführt. Ist ein TAB-Wert kleiner als die aktuelle Cursorposition, wird die Anweisung erst in der nächsten Zeile wirksam.

TAB() ist nur im Zusammenhang mit **PRINT** einsetzbar (z.B. `PRINT ''Text<'';TAB(2);''>Text''`).

3.3. TEXTBILDSCHIRM - STEUERUNG

SCROLL OFF { SCRO OFF } — Text-Scrolling ausschalten

```
SCROLL OFF
```



Unterdrückt das Text-Scrolling bei Erreichen der letzen Bildschirm-Textposition (s. **SCROLL ON**).

SCROLL ON { SCRO ON } — Text-Scrolling anschalten

```
SCROLL ON
```



Schaltet - auch im Grafikmodus - das Aufwärtsscrolling des Bildschirms an. Wird an der letzten Cursorposition in der rechten unteren Ecke des aktuellen Bildschirms (s. auch **TCLIP**) Text ausgegeben und die aktuelle Textausgabe will noch weitere Zeichen ausgeben, so wird der Bildschirm-Inhalt um die aktuelle Textzeilen-Höhe nach oben verschoben und die Textausgabe am linken Rand der untersten (nun wieder leeren) Bildschirm-Textzeile fortgesetzt.

Default-Einstellung bei Programmstart ist **SCROLL ON**.

_TS — Textbildschirm-Adresse liefern

```
Var=_TS
```



_TS ist eine reservierte Variable, die konstant die Segmentadresse des aktuellen Textbildschirms enthält (z.B. `POKE _TS:0,65` schreibt im Textmodus ein 'A' in die linke obere Ecke des Bildschirms).

TBOX {TB} — Rechteck im Textmodus zeichnen

```
TBOX Modus,Xstart,Ystart,Breit,Hoch
```



TBOX zeichnet im Textmodus beliebige Rechtecke. Dazu werden die im IBM-Zeichensatz enthaltenen Rahmen-Zeichen (ASCII 179 - 218) verwendet. Die Attribut-Einstellungen durch **TCOLOR** werden dabei berücksichtigt.

'Xstart' gibt die Textraster-Spalte (1 bis 79 je nach Auflösung) und ***'Ystart'*** die Textraster-Zeile (1 bis 24 je nach Auflösung) für die obere, linke Ecke des Rahmens an. Durch die Parameter ***'Breit'*** und ***'Hoch'*** wird die Ausdehnung des Rahmens in die jeweilige Richtung angegeben.

'Modus' bestimmt die Rahmen-Gestaltung:

0 = Leerzeichen-Rahmen in Hintergrundfarbe
1 = einfacher Rahmen
2 = doppelter Rahmen

TCOLOR {TCO} — Textattribut im Textmodus bestimmen

```
TCOLOR Attribut
```



Im Textmodus kann durch TCOLOR die Textfarbe und das Hintergrund-*'Verhalten'* gesteuert werden. Der ***'Attribut'***-Parameter setzt sich aus zwei sogenannten *'Nibbels'* zusammen. Ein Nibbel ist ein 4Bit-Wert von 0 bis 15, der bequem mit nur einer Hexadezimalziffer (0 bis F) dargestellt werden kann. In Kenntnis dieses Umstands ist es daher aus Gründen der Übersichtlichkeit ratsam, diesen Parameter hier auch als Hexadezimalwert anzugeben (z.B. `TCOLOR $4E`).

Bei Verwendung einer MDA- oder Hercules-Adapters ergibt sich dieser ***'Attribut'***-Wert wie folgt:

1.Nibbel (Hintergrund):		2.Nibbel (Vordergrund):	
0 =	schwarz	0 =	schwarzes Zeichen
7 =	weiß	1 =	schwarz+Unterstrich
		2 =	weißes Zeichen
		3 =	weiß+Unterstrich
Durch Addition d.Wertes könne blinkende Zeichen erzeugt werden		Durch Addition des Wertes 8 kann die Helligkeit des Zeichens erhöht werden	
z.B. `TCOLOR $F9:` (15*16+9 = 249 dez.)		helles schwarzes Zeichen auf blinkendem weißen Hintergrund	

Bei **CGA-**, **EGA-** und **VGA-**Karten ergibt sich folgende Tabelle:

1.Nibbel (Hintergrund):		2.Nibbel (Vordergrund) =Zeichenfarbe:	
0 ($0)	Schwarz	0 ($0)	Schwarz
1 ($1)	Blau	1 ($1)	Blau
2 ($2)	Grün	2 ($2)	Grün
3 ($3)	Türkis	3 ($3)	Türkis
4 ($4)	Rot	4 ($4)	Rot
5 ($5)	Purpur	5 ($5)	Purpur
6 ($6)	Braun	6 ($6)	Braun
7 ($7)	Hellgrau	7 ($7)	Hellgrau
		8 ($8)	Dunkelgrau
Durch Addition des Wertes 8 können blinkende Zeichen erzeugt werden		9 ($9)	Hellblau
		10 ($A)	Hellgrün
		11 ($B)	Helltürkis
		12 ($C)	Hellrot
		13 ($D)	Hellpurpur
		14 ($E)	Gelb
		15 ($F)	Weiß
z.B. `TCOLOR $FF:` (15*16+15 = 255 dez.)		grellweißes Zeichen blinkend auf hellgrauem Hintergrund	

TGET { TGE } Bildschirmbereich im Textmodus speichern

```
TGET Tx_links,Ty_oben,Tx_rechts,Ty_unten,Var$
```



Dieser Befehl ist nur im Textmodus einsetzbar. Durch ***'Tx_links'***, ***'Ty_oben'*** und ***'Tx_rechts'***, ***'Ty_unten'*** wird im aktuellen Textraster ein Ausschnitt des Textbildschirms definiert, welcher als ASCII-Muster in die Stringvariable ***'Var$'*** eingelesen wird.

'Var$' enthält anschließend (jeweils im HI-Byte) im ersten Wort die Breite des Ausschnitts, im zweiten Wort die Höhe sowie im dritten Wort $FFFF. Daran schließen sich der Reihe nach im Wortformat die Textinformationen an. Für jedes Zeichen auf dem Bildschirm wird zuerst ein Byte ASCII-Information und dann ein Byte Farb-Information gespeichert. Dabei wird hinter dem Header zuerst die erste Ausschnittzeile abgelegt, danach die zweite, dann die dritte usw.

Ein Textausschnitt mit 5 Zeichen Breite und 3 Zeichen Höhe hat demnach eine Länge von 3 Header-Words plus 5 mal 3 Daten-Words = 18 Words = 36 Byte.

TCLIP { TC } Textausgabe-Bereich bestimmen

```
TCLIP Tx_links,Ty_oben,Tx_rechts,Ty_unten
TCLIP OFF
```



Dieser Befehl ist nur im Textmodus einsetzbar. Er bestimmt einen Bildschirm-Ausschnitt, der bis zum nächsten TCLIP bzw. TCLIP OFF als Ausgabe-Bildschirm angesehen wird. **WRAP ON** und **WRAP OFF** bzw. **SCROLL ON** und **SCROLL OFF** wirken sich auf diesen Bereich genauso aus, wie dies beim Gesamt-Bildschirm der Fall wäre. Der Befehl ist also nicht ganz mit dem Grafik-Clipping (s. **CLIP**) vergleichbar.

Die zweite Variante TCLIP OFF hebt den gfls. durch TCLIP definierten Bildschirm-Bereich auf und bewirkt wieder die Text-Ausgabe auf dem Gesamt-Bildschirm. TCLIP OFF wird bei Programmende nicht automatisch ausgeführt. Das zuletzt aktive TCLIP-Textfenster kann also beim nächsten Programmstart noch gültig sein.

TPBOX { TPB } gefüllt.Rechteck im Textmodus zeichnen

```
TPBOX Modus,Xstart,Ystart,Breit,Hoch
```



TPBOX zeichnet im Textmodus mit der Hintergrundfarbe gefüllte Rechtecke. Weitere Erläuterungen zu den Parametern finden Sie unter **TBOX**.

TPUT { TPU } Bildschirmbereich im Textmodus setzen

```
TPUT Tx_links,Ty_oben,Var$
```

Dieser Befehl ist nur im Textmodus einsetzbar. Durch das Koordinatenpaar *'Tx_links'* und *'Ty_oben'* wird eine Position im aktuellen Textraster bestimmt, an welcher die linke, obere Ecke eines durch **TGET** in der Stringvariablen *'Var$'* gespeicherten Bildschirmausschnitts angelegt wird. Dort wird der Inhalt von *'Var$'* als Textrechteck wieder ausgegeben.

TTEXT { TT } Text auf Monochrom-Karte ausgeben

```
TTEXT Spalte,Zeile,''Text''
TTEXT Spalte,Zeile+Modus*256,''Text''
```



Wenn Sie über einen Zweitmonitor verfügen und diesen gfls. über eine Monochrom-Karte parallel betreiben, haben Sie mit diesem Befehl die Möglichkeit, *'Text'* gezielt darauf auszugeben. Dabei gibt *'Spalte'* die Cursorspalte (1 - 80) und *'Zeile'* die Cursorzeile (1 - 25) an, in welcher die Ausgabe beginnen soll. Wird zum Parameter 'Zeile' ein 'Modus'-Wert*256 addiert, kann damit gleichzeitig der Ausgabemodus für den *'Text'* bestimmt werden (s. **TCOLOR**).

WRAP ON { WR ON } Zeilen-Umbruch anschalten

```
WRAP ON
```



Schaltet - auch im Grafikmodus - das Abknicken einer Textausgabe bei Erreichen des rechten Randes des aktuellen Textausgabe-Bereichs an. Die Ausgabe wird am linken Rand der nächsten Zeile fortgesetzt.

Default-Einstellung ist bei Programmstart immer WRAP ON.

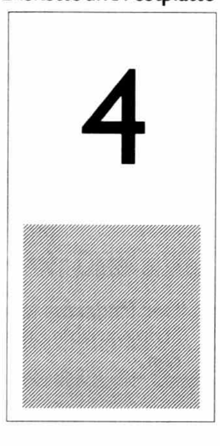
WRAP OFF { **WR OFF** } Zeilen-Umbruch ausschalten

```
WRAP OFF
```

Unterdrückt das Abknicken der Textausgabe bei Erreichen der letzen Text-X-Position am rechten Rand. Die Ausgabe läuft - unsichtbar - über den rechten Ausschnittrand hinaus (s. **WRAP ON**).

Notizen:

Diskette und Festplatte



4. DISKETTE UND FESTPLATTE

Bei allen Diskettenoperationen, denen ein Dateiname zu übergeben ist, besteht die Möglichkeit, mit diesem einen *Suchpfad* zu definieren, über den der Dateizugriff ausgeführt werden soll:

- Laufwerkbezeichnung:
  Buchstabe mit nachfolgendem Doppelpunkt
  z.B.: 'A:' für das erste Laufwerk, 'B:' für das Zweite, etc....

- Fragezeichen innerhalb des Dateinamens sind Platzhalter oder auch sog. *'Wildcards'*. Es werden alle Dateien angesprochen, die bis auf die Fragezeichen mit dem angegebenen Dateinamen übereinstimmen.

  z.B.: `''C:\GFA\PROGRAM?.GF?''`

  wäre sowohl für:

  `''C:\GFA\PROGRAMM.GFA''`

  als auch für:

  `''C:\GFA\PROGRAMS.GFW''`

  zutreffend.

- Sternchen im Dateinamen sind Platzhalter für einen ganzen Bereich (vor oder nach dem Trennpunkt).

  z.B.: `''C:\GFA\*.GFA''`

  wäre sowohl für:

  `''C:\GFA\PROG1.GFA''`

  als auch für:

  `''C:\GFA\PROG2.GFA''`

  zutreffend. Also für alle Dateien im angegebenen Pfad, die über die *Extension* ".GFA" verfügen (extension= engl.; *'Erweiterung'*)

Der Pfadname kann in den meisten Fällen als Stringausdruck, als Stringvariable oder als Kombination von beidem übergeben werden.

Im Folgenden taucht häufig der Begriff **'#Kanal'** auf. Damit ist bei dateirelativen Diskettenoperationen der *Identifikator* (0-99) der jeweils angesprochenen Datei gemeint.

4.1. BLOCK - OPERATIONEN

BLOAD { BL } — Datei in Speicherbereich laden

```
BLOAD ''Dateiname'',Ziel
```

'Dateiname' enthält den Namen der zu ladenden Datei. Der Parameter ***'Ziel'*** bestimmt die Startadresse des Zielbereichs.

BSAVE { BS } — Speicherbereich auf Datenträger sichern

```
BSAVE ''Dateiname'',Quell,Bytes
```

'Dateiname' enthält den Namen der Datei, in welcher der Speicherbereich abgelegt wird. ***'Quell'*** enthält die Startadresse (Segment:Offset) und ***'Bytes'*** die Bytegröße des Bereichs.

4.2. UMBENENNEN, LÖSCHEN, NACHFRAGEN UND SUCHEN

DFREE() — freien Disk- bzw. Harddisk-Speicher ermitteln

```
Var=DFREE(Laufwerk)
```

'Laufwerk' steht für die Nummer des Laufwerkes, dessen freier Speicher ermittelt werden soll.

0 = akt. Laufw. / 1 = Laufw. A / ... 26 = Laufw. Z

EXIST() — Existenz einer Datei prüfen

```
Var=EXIST(Dateiname)
```

'Dateiname' enthält den *Suchpfad* (s. Kapitelanfang) der Datei, deren Vorhandensein überprüft werden soll.

Liefert	0	(FALSE	=Datei nicht vorhanden)
oder	-1	(TRUE	=Datei vorhanden).

FGETDTA() Adresse der 'Disk-Transfer-Area' ermitteln

```
Var=FGETDTA()
```

Liefert die Startadresse der mind. 42 Byte großen *'Disk-Transfer-Area'* im MSDOS-Adressformat. Das HI-Word des gelieferten Longwords stellt dabei die Segmentadresse und das LO-Word den Offset dar.

Verschiedenen Diskettenzugriffe (**DIR, FILES, EXIST, FSFIRST** etc.) verwenden die *DTA* (üblicherweise bei **_PSP**+128) zur Ablage des aktuell gelesenen Dateinamens und seiner Attribute.

Offset: **Bedeutung:**

0 - 20 für MSDOS reserviert (21 Bytes)
21 Datei-Attribut (1 Byte)

```
BYTE{FGETDTA()+21}
```

ergibt:

0	=	normale Datei (lesen/schreiben)
1	=	schreibgeschützte Datei
2	=	versteckte Datei
4	=	System-Datei
8	=	Disketten-Name (Volume Label)
16	=	Unter-Verzeichnis (Ordner)
32	=	Archiv-Datei (nur bei Harddisk)

22 Uhrzeit
(2 Byte: 5BitStd+6BitMin+5BitSec)
24 Datum
(2 Byte: 7BitTag+4BitMon+5BitJhr+1980)
26 Dateilänge (4 Byte)
30 - 41 Dateiname (12 Byte)

FSETDTA() Adresse der 'Disk-Transfer-Area' bestimmen

```
VOID FSETDTA(Adresse)
```

Ermöglicht die Bestimmung der Startadresse der *'Disk-Transfer-Area'* (s. **FGETDTA()**). ***'Adresse'*** ist eine beliebige gerade

Adresse, die mindestens 128 Byte vor Ende eines Speicher-Segments liegen sollte.

FSFIRST() — Datei suchen

```
Var=FSFIRST(Pfad$,Attribut)
```

Sucht die erste vorkommende Datei, auf welche die Suchvorgabe ***'Pfad$'*** zutrifft (s. Kapitelanfang). Zusätzlich kann ein Datei-Attribut (s. **FGETDTA()**) angegeben werden. Besitzt eine Datei nicht dieses Attribut, wird es bei der Suche ignoriert.

Wird eine Datei gefunden, können die entsprechenden Daten aus der *'Disk-Transfer-Area'* ausgelesen werden (s. **FGETDTA()**). Ist keine Datei mit den angegebenen Kriterien zu finden, wird in ***'Var'*** ein negativer Wert geliefert.

FSNEXT() — weitere Datei suchen

```
Var=FSNEXT()
```

Sucht die nächste Datei, auf welche die beim letzten **FSFIRST()** vorgegebenen Suchkriterien zutreffen. Wird eine Datei gefunden, können die entsprechenden Daten aus der *'Disk-Transfer-Area'* ausgelesen werden (s. **FGETDTA()**). Ist keine weitere Datei mit den angegebenen Kriterien mehr zu finden, wird in ***'Var'*** ein negativer Wert geliefert.

KILL { KI } — Disk-Datei löschen

```
KILL ''Dateiname''
```

'Dateiname' bestimmt als Textausdruck oder als Stringvariable den Namen einer Datei, die gelöscht werden soll.

NAME..AS { NA..AS } — Datei umbenenennen

```
NAME ''Name_alt'' AS ''Name_neu''
```

'Name_alt' ist der Dateiname, der ersetzt werden soll. Nach einem angehängten ***'AS'*** wird ***'Name_neu'*** übergeben, der dann ***'Name_alt'*** ersetzt. Innerhalb einer Station kann eine im

Hauptdirectory verzeichnete Datei durch eine entsprechende Pfadangabe im neuen Namen in ein vorhandenes Unter-Verzeichnis (Ordner) verlegt werden.

RENAME..AS { REN..AS } Datei umbenennen

```
RENAME ''Name_alt'' AS ''Name_neu''
```

Entspricht exakt dem Befehl **NAME** (weiteres s. dort).

4.3. INTERPRETER - BEFEHLE

LIST { LIS } Programm als ASCII-Code listen/speichern

```
LIST [''Programmname'']
```

LIST ohne Zusätze gibt den aktuellen Programmtext auf dem Monitor aus. Ein Abbruch des Listings kann durch die Break-Funktion (s. **ON BREAK**...) erfolgen.

Das Listing wird als *ASCII*-File auf Diskette gespeichert, wenn dem Befehl ein ***'Programmname'*** übergeben wird. Befindet sich bereits ein Programm gleichen Namens auf der Diskette, wird es automatisch in ***'Programmname.BAK'*** umbenannt.

Hiermit abgespeicherte Programme können mit der Editor-Funktion *'Merge'* in den Arbeitsspeicher eingelesen werden.

LOAD { LOA } Programm in Arbeitsspeicher laden

```
LOAD ''Programmname''
```

'Programmname' ist der Name des zu ladenden Programms. Das aktuelle Programm wird beendet und aus dem Speicher gelöscht. Die voreingestellte Datei-Extension ist *'.GFA'*.

PSAVE { **PSA** } Programm speichern (listgeschützt)

```
PSAVE ''Programmname''
```



Für diesen Befehl gilt die gleiche Ausführung wie zu **SAVE**. PSAVE-Programme werden nach dem Laden selbsttätig gestartet und können anschließend im Editor nicht gelistet werden.

SAVE { **SA** } Programm speichern (Token-Code)

```
SAVE ''Programmname''
```



Speichert das im Arbeitsspeicher befindliche Programm GFA-*tokencodiert* unter ***'Programmname'*** auf Diskette. Die voreingestellte Datei-Extension ist *'.GFA'*. Fehlt die Extension bei der Angabe des Namens, wird der ***'Programmname'*** automatisch auf ***'Programmname.GFA'*** ergänzt.

Befindet sich bereits ein Programm gleichen Namens auf der Diskette, wird es automatisch in ***'Programmname.BAK'*** umbenannt.

4.4. DIRECTORY - OPERATIONEN

CHDIR { **CHD** } Ordner wechseln

```
CHDIR ''Ordner''
```

Ist nur innerhalb eines Laufwerks einsetzbar. Ein Laufwerkswechsel ist durch CHDIR nicht möglich (s. **CHDRIVE**). Besteht ***'Ordner'*** nur aus dem *Backslash* ("\"), wird danach nicht auf ein Unter-Verzeichnis (*Ordner*), sondern auf das Haupt-Directory (*Wurzel-Verzeichnis*) zugegriffen.

Es kann statt eines gfls. übergeordneten Verzeichnisnamens das Kürzel **".."** verwendet werden, wodurch ein Wechsel aus dem aktuellen Verzeichnis in den Ordner der darüberliegenden Ebene erfolgt. Ist kein Über-Verzeichnis vorhanden, erscheint eine entsprechende Fehlermeldung.

CHDRIVE { CHDR } aktuelles Laufwerk bestimmen

```
CHDRIVE Laufwerk CHDRIVE Pfad$
```



'Laufwerk' bestimmt das aktuelle Laufwerk:

1	=	Laufwerk A:
2	=	Laufwerk B:
.. 26	=	Laufwerk Z:

Es kann auch ein Textausdruck angegeben werden, dessen erstes Zeichen als Stationsbestimmung interpretiert wird

z.B. `CHDRIVE ''A:\*.*''`

DIR Directory ausgeben

```
DIR [''Pfad''] [TO ''Datei'']
```

Das Disk-, bzw. Verzeichnis-Directory kann auf dem Bildschirm (`DIR [''Pfad''])` oder mit dem Zusatz `''Pfad'' TO ''Datei''` in eine Diskdatei oder eine virtuelle Datei (z.B. `''CON:''` oder `''PRT:''`) ausgegeben werden.

Wird die Ausgabe aller Dateien gewünscht, wird üblicherweise `''A:\*.*''` bzw. `''Verzeich\*.*''` etc. als ***'Pfad'*** verwendet. Endet ***'Pfad'*** mit `':'` oder `'\'`, so wird `'*.*'` vom Interpreter intern automatisch hinzugefügt.

DIR$() aktuellen Ordnernamen ermitteln

```
Var$=DIR$(Laufwerk)
```



'Laufwerk' bestimmt die zu überprüfende Station:

0	=	aktuelles Laufwerk
1	=	Laufw. A
.. 26	=	Laufw. Z

Ist z. Zt. kein Unterverzeichnis (*Ordner*) geöffnet, so wird ein Leerstring zurückgegeben.

FILES { FI } Directory (erweitert) ausgeben

```
FILES [''Pfad''] [TO ''Datei'']
```



Erstellt das Inhaltsverzeichnis einer Diskettenstation oder eines Unterverzeichnisses zusammen mit den Angaben über Länge, Uhrzeit und Datum der einzelnen Dateien. Weitere Erläuterungen finden Sie unter **DIR**.

MKDIR { MK } Ordner erzeugen

```
MKDIR ''Ordner''
```



Erzeugt ein Unterverzeichnis (hier mit der Bezeichnung ***'Ordner'***) im aktuellen Directory. Durch eine entsprechende Pfadangabe kann der Ordner auch in beliebigen Verzeichnissen erzeugt werden (z.B. `''C:\Lager_1\Fach_3''` erzeugt den Ordner *'Fach_3'* innerhalb des Ordners *'Lager_1'*)

RMDIR { RM } Ordner löschen

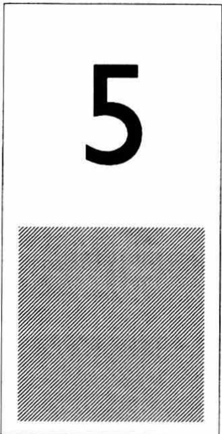
```
RMDIR ''Ordner''
```



'Ordner' enthält den Namen (gfls. incl. Pfad) des Unterverzeichnisses, das gelöscht werden soll. Das Verzeichnis darf dazu keine Dateien oder Unterordner mehr enthalten (auf versteckte Dateien achten!).

Notizen:

Datei-Handhabung



5. DATEI - HANDHABUNG

5.1. ÖFFNEN, SCHLIESSEN, POSITION UND DATUM

CLOSE {CL} Datenkanal schließen

```
CLOSE [#Kanal]
```



Wird der Befehl ohne die Option ***'#Kanal'*** verwendet, werden alle offenen Dateien geschlossen, sonst nur die mit der angegebenen Kanalnummer, sofern sie vorher mit **OPEN** geöffnet wurde. Bei Programmende werden alle offenen Dateien automatisch geschlossen.

OPEN {O} Datenkanal öffnen

```
OPEN ''Modus'',#Kanal,''Dateiname'' [,Satzlänge]
```



Es können Datenkanäle (Festspeicher-Dateien, Peripherie-Geräte) zum Einrichten, Schreiben, Lesen, Ergänzen oder Erneuern geöffnet werden.

'Modus':

"O" = (*Output*) - öffnet eine Datei zum Schreiben von Daten, oder, falls die Datei noch nicht vorhanden ist, installiert diese mit der angegebenen Kanalnummer. Existiert bereits eine Datei mit derselben Namensspezifikation, wird sie durch die neue Datei überschrieben, bzw. gelöscht.

"I" = (*Input*) - öffnet eine bestehende Datei zum Lesen der darin enthaltenen Daten.

"A" = (*Append*) - öffnet eine bestehende Datei und setzt den Filepointer hinter das letzte Byte der Datei. Alle an diese Datei auszugebenden Daten werden an dies Dateiende angehängt.

"U" = (*Update*) - öffnet eine bestehende Datei mit der Möglichkeit, diese gleichzeitig zu lesen und in sie zu schreiben.

"R" = (*Random*) - öffnet eine *Random-Access-Datei* zum Lesen und Schreiben.

Mit *Random-Access-Dateien* ist es möglich, Datensätze für wahlfreien Zugriff innerhalb einer Datei anzulegen. Maximal sind 65535 Datensätze pro Datei möglich.

Diese Datensätze werden durch den Befehl **GET#** gelesen und durch **PUT#** geschrieben. Die Einrichtung der Datei erfolgt durch den Befehl **FIELD#**.

Der OPEN-Befehl erweitert sich gegebenenfalls hierfür um den optionalen Parameter ***'Satzlänge'***. Wird er verwendet, gibt er die Anzahl der Bytes an, die den Datensätzen zugeordnet werden soll. Wenn nicht, sieht der Interpreter automatisch eine Satzlänge von 128 Byte vor.

Die Länge einer *'R'*-Datei hängt von der eingerichteten Satzlänge und der Anzahl der Datensätze ab (Satzlänge*Satzanzahl).

'#Kanal' gibt den Datei-*Identifikator* (0-99) an.

'Dateiname' beschreibt den *Zugriffspfad (Suchpfad)* der zu bearbeitenden Datei.

Außer Festspeicher-Dateien können auch gfls. angeschlossene Periperiegeräte angesprochen werden:

"CON:"	= Consolport (Monitor ohne Steuerzeichen)
"VID:"	= Video-Port (Monitor incl. Steuerzeichen)
"LPT1:...LPT4:"	= Parallel-Ports (LPT1 = Drucker-Port)
"COM1:...COM4:"	= Serielle Ports (COMmunication-Ports)
"MON:"	= Ausgabe auf Monochrom-Zweitmonitor mittels MDA/HGC-Karte (z.B. per **TRON #**)

Zu ***'VID:'*** beachten Sie bitte das Beispiel zu **XLATE$()**. ***'Modus'*** kann bei Verwendung dieser Möglichkeit entfallen.

Die Eingabe der OPEN-Zeile kann extrem verkürzt werden. So wird z.B.

```
o I 1 Name.Lst
```

vom Editor in

```
OPEN ''I'',#1,''Name.Lst''
```

und z.B.

```
o o 1 Name$
```

in

```
OPEN ''o'',#1,Name$
```

umgewandelt. Alle An- und Ausführungsstriche, das Nummernzeichen, sowie die Trennkommas können vernachlässigt werden, sofern zwischen den einzelnen Komponenten ein Leerzeichen angegeben wird.

RELSEEK { REL } — Filepointer verschieben

```
RELSEEK #Kanal,[-] Offset
```

'Offset' enthält die Anzahl der Bytes, um die der *Filepointer* ohne sonstige Aktivitäten verschoben werden soll. Negative Werte bewirken eine Verschiebung von der aktuellen Position in Richtung Dateianfang.

SEEK { SE } — Filepointer setzen

```
SEEK #Kanal,[-] Position
```

Der Parameter ***'Position'*** gibt das Byte an, auf welches der zu ***'#Kanal'*** gehörige *Filepointer* absolut gesetzt werden soll. Negative Werte bewirken, daß der Byteoffset vom Dateiende aus gezählt wird.

TOUCH { TOU } — Datei-Zeitangabe ändern

```
TOUCH [#]Kanal
```

Schreibt die aktuelle System-Zeitangabe (s. **TIME$**) in die dafür vorgesehene Position des Datei-Eintrags im betreffenden Directory. Kurz: aktualisiert den Zeiteintrag der Datei ***'Kanal'*** (0-99). Dieser muß durch **OPEN** vorher geöffnet worden sein.



5.2. DATEI - READ/WRITE

BGET { BG } — Teildatei lesen

```
BGET #Kanal,Ziel,Anzahl
```



Ab aktueller Filepointerposition werden ***'Anzahl'*** Bytes der Datei in den Speicher gelesen. ***'Ziel'*** beschreibt die Startadresse (Segment:Offset) des Ziel-Speicherbereichs.

BPUT { BP } — Teildatei schreiben

```
BPUT #Kanal,Quell,Anzahl
```



Ab der Startadresse ***'Quell'*** (Segment:Offset) werden ***'Anzahl'*** Bytes aus dem Speicher gelesen und die Datei ab der aktuellen Filepointerposition mit diesen Daten überschrieben.

INP(#) — Daten byteweise aus Datei lesen

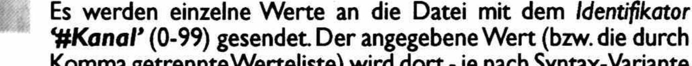
```
Var=INP(#Kanal)
```



'#Kanal' beschreibt den *Datei-Identifikator* (0-99), aus dem ein einzelner Bytewert an der aktuellen *Filepointer*-Position gelesen werden soll. Der gelesene Bytewert steht anschließend in ***'Var'***.

OUT # { OU } — Daten einzeln in Datei schreiben

```
OUT #Kanal,Byte1 [,Byte2 [,...]]
OUT& #Kanal,Word1 [,Word2 [,...]]
OUT% #Kanal,Long1 [,Long2 [,...]]
```



Es werden einzelne Werte an die Datei mit dem *Identifikator* ***'#Kanal'*** (0-99) gesendet. Der angegebene Wert (bzw. die durch Komma getrennte Werteliste) wird dort - je nach Syntax-Variante als Byte, Word oder Long - an die aktuelle Filepointerposition geschrieben.

PRINT # — Daten in Datei ausgeben

```
PRINT #Kanal,[;] ''Text''[[;,']Var[;,']Expr...;]
```

Siehe Beschreibung zu **PRINT**. Bei Verwendung dieser Syntaxform werden die Daten an der aktuellen *Filepointer*-Position der geöffneten (s. **OPEN**) und durch ***'Kanal'*** angegebenen Datei geschrieben.

PRINT # USING — Formatierte Ausgabe in Datei

```
PRINT #Kanal, USING ''format''[,;']Expr[,Var,..]
```

Siehe Beschreibung zu **PRINT USING**. Dieser Befehl bewirkt die formatierte Datenausgabe in die durch ***'Kanal'*** angegebene Datei (0-99).

RECALL { REC } — Stringfeld aus Datei lesen

```
RECALL [#]Kanal,Feld$(),Anzahl,Zeilen
```

Liest ***'Anzahl'*** Textzeilen aus der Datei mit der Kennung ***'Kanal'*** (0-99; muß durch **OPEN** geöffnet worden sein) in das - vorher ausreichend zu dimensionierende - Stringfeld ***'Feld$()'***. Ein *'Carriage Return'* (**CHR$(13)**) wird dabei als Zeilenende interpretiert.

Den Feldelementen wird der Reihe nach je eine Zeile zugeordnet. Ist das Feld zu kurz (falls Elementeanzahl kleiner als die Anzahl der gelesenen Zeilen), wird der Leseprozess beim letzten Element abgebrochen. Stößt das Programm beim Lesen auf das Dateiende, wird ebenfalls abgebrochen (ohne Fehlermeldung). In der Rückgabevariablen ***'Zeilen'*** (Fließkomma- oder 4Byte-Integervariable) steht anschließend die Anzahl der tatsächlich gelesenen Zeilen.

STORE { ST } — Stringfeld in Datei ablegen

```
STORE [#]Kanal,Feld$()[,Anzahl]
```

Speichert die Elemente des Stringfeldes ***'Feld$()'*** der Reihe nach in der geöffnet Datei ***'Kanal'***. Als Endmarkierung wird dort jeder geschriebenen Zeile ein *CR/LF* (*'Carriage Return'/'Line Feed'* = **CHR$(13)/CHR$(10)**) angehängt.

Durch den optionalen Parameter ***'Anzahl'*** kann bei Bedarf die Anzahl der auszugebenden Elemente begrenzt werden.

5.3. DATEI - INFO

_FILE() — MSDOS-Handle einer Datei liefern

```
Var=_FILE(Kanal)
```



Liefert das MSDOS-Systemhandle der durch ***'Kanal'*** (s. **OPEN**:0-99) angegeben (und geöffneten) Datei. Ist kein Kanal geöffnet, wird eine Null zurückgegeben.

Die Peripherie-*'Dateien'* sind dabei an folgenden Handles zu erkennen:

CON:	=	-1
VID:	=	-2
LPT1: bis *LPT4:*	=	-3 bis -6
COM1: bis *COM4:*	=	-7 bis -10
MON:	=	-11

Anhand dieser Handles ist es möglich, auch Peripheriegeräte aus dem GFA-BASIC heraus direkt über die entsprechenden DOS-Funktionen anzusprechen.

EOF() — Datei auf Dateiende prüfen

```
Var=EOF(#Kanal)
```



Befindet sich der *Filepointer* auf dem letzten Byte der angegebenen und geöffneten Datei, wird -1 geliefert, andernfalls 0.

LOC() — Filepointerposition liefern

```
Var=LOC(#Kanal)
```



Der zu jeder geöffneten Datei gehörige Schreib- und Lesezeiger (*Filepointer*) 'zeigt' auf die aktuelle Position des Schreib- und Lesekopfes innerhalb der Datei. Diese Position wird durch LOC() geliefert.

LOF() Dateilänge ermitteln

```
Var=LOF(#Kanal)
```



Es wird die Größe der mit **OPEN** geöffneten Datei mit dem angegebenen *Identifikator* (0-99) ermittelt.

5.4. RANDOM-ACCESS-OPERATIONEN

FIELD {FIE} Datensatz in Elemente unterteilen

```
FIELD #Kanal,Anz AS Var1$ [,Anz AS Var2$,...]
FIELD #Kanal,Anz AT(Adr1) [,Anz AT(Adr2),...]
```



Ordnet in der ersten Syntax-Variante den einzelnen Datensatz-Elementen einzelne Bytelängen zu und füllt die dazugehörigen Aufnahme-Stringvariable(n) ***'Var'*** mit Leerzeichen.

'#Kanal' gibt die *'R'*-Datei an (s. **'Open'**), die in Datenfelder aufgeteilt werden soll. Es wird jeweils die in ***'Anz'*** angegebene Anzahl an Bytes der nach ***AS*** folgenden Variablen zugeordnet. Die mit

```
'Open,''R'',#Kanal,''Dateiname'',Satzlänge'
```

eingerichtete *Satzlänge* wird so in verschiedene Datenfelder eingeteilt. Die Summe aller Feldgrößen muß der angegebenen Satzlänge entsprechen. Für jeden geöffneten Datenkanal im *'R'*-Modus ist nur eine Field-Anweisung zulässig.

Bei der ***'AS'***-Variante müssen gfls. numerische Daten durch **MKD$/MKI$/MKL$/MKS$** vor ihrer Speicherung in das Stringformat konvertiert und die durch **GET#** gelesenen Daten wieder durch **CVD/CVI/CVL/CVS** in das entsprechende numerische Format zurückgewandelt werden.

Bei der zweiten Syntax-Variante ist dies nicht notwendig, da als Quell- und Zieladresse ja die Startadresse einer entsprechenden numerischen Variable angegeben werden kann. Dabei enthält ***'Anz'*** die Anzahl an Bytes, die ab der zugehörigen - hinter ***AT*** in Klammern gesetzten - Adresse gelesen werden sollen.

z.B.:

```
a%=673123        --
b%=VARPTR(a%)     |-     Werte
c&=1000           |- vorbereiten
d=61234.1231     --
FIELD #1,4 AT(b%),2 AT(*c&)  1. FIELD-Zeile
FIELD #1,8 AT(V:d)       gfls.2. FIELD-Zeile
```

Textstrings sollten nicht mit der ***'AT'***-Variante verwaltet werden, da sich die Anfangsadresse einer Textvariablen nicht unbedingt statisch verhält.

'AT'- und ***'AS'***-Elemente können in einer FIELD-Zeile beliebig gemischt werden (z.B. `FIELD #1,20 AS a$,2 AT(*b&),...`). Außerdem kann die FIELD-Aufteilung für eine Datei auf mehrere Programmzeilen verteilt sein.

GET # { GE } — Datensatz lesen

```
GET [#]Kanal [,Satznummer]
```

'Kanal' gibt die *'R'*-Datei (s. **OPEN**) an, aus welcher der angegebene Datensatz gelesen werden soll. Bei der Zuordnung von Datensätzen zu einer *'R'*-Datei ist jedem Satz eine Nummer zuzuteilen. Unter Angabe dieser ***'Satznummer'*** (max. 65535) kann der Datensatz mit GET# wieder in die mit **FIELD** spezifizierten Stringvariablen bzw. Speicherbereiche zurückgelesen werden. Fehlt ***'Satznummer'***, wird jeweils der nächste Datensatz gelesen.

PUT # { PU } — Datensatz schreiben

```
PUT [#]Kanal [,Satznummer]
```

Es wird der mit ***'Satznummer'*** (max. 65535) definierte Datensatz aus den mit **FIELD** spezifizierten Stringvariablen bzw. Speicherbereichen in eine *'R'*-Datei mit der Nummer ***'Kanal'*** geschrieben. Fehlt ***'Satznummer'***, wird der jeweils nächste Datensatz geschrieben.

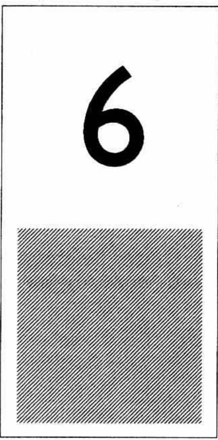
RECORD { RECO } GET#/PUT#-Satzzeiger positionieren

```
RECORD [#]Kanal,Satznummer
```

Setzt den *Satz-Zeiger* einer *'R'*-Datei für den nächsten **GET#**- oder **PUT#**-Zugriff auf den durch ***'Satznummer'*** spezifizierten Datensatz. Wird beim nächsten **GET#** oder **PUT#** der Parameter ***'Satznummer'*** ausgelassen, wird der durch RECORD hier angegebene Datensatz als nächstes gelesen bzw. geschrieben.

Notizen:

Peripherie



6

6. PERIPHERIE

6.1. HARDWARE - I/O

INP(PORT) Daten aus Hardware-Port lesen

```
Bytevar=INP(PORT Nr)
Bytevar=INP|(PORT Nr)
Wordvar=INP&(PORT Nr)
Longvar=INP%(PORT Nr)
```


Aus dem Hardware-*Port* ***'Nr'*** (gfls. Peripherie-Chip-Adresse) wird ein Wert gelesen.

Der ***'Nr'***-Index für die in Frage kommenden Adreßbereiche ist nicht explizit definiert und der Befehl nur bei genauester Kenntnis der Hardware-Spezifikationen einsetzbar.

OUT { **OU PORT** } Daten in Hardware-Port schreiben

```
OUT PORT Nr,Bytewert
OUT| PORT Nr,Bytewert
OUT& PORT Nr,Wortwert
OUT% PORT Nr,Longwert
```


Der angegeben Wert wird in den Hardware-Port ***'Nr'*** geschrieben. Weiteres siehe unter **INP(PORT)**.

6.2. DRUCKER-ANWEISUNGEN

HARDCOPY { **HA** } Text-Bildschirm auf Drucker ausgeben

```
HARDCOPY
```


Dieser Befehl ist mit einem Druck auf die *'Druck'*-Taste identisch. Ist der Drucker nicht empfangsbereit, ertönt ein Signalton.

LLIST { LL } — Programmlisting ausdrucken

```
LLIST [''Dateiname'']
```



Das Druckerlisting kann - falls die Break-Funktion (s. **ON BREAK...**) aktiv ist - jederzeit durch die Breaktasten unterbrochen werden. Der Interpreter ist dann wieder arbeitsbereit und es wird nur noch der Inhalt des Druckerpuffers ausgedruckt. Soll das auch noch unterbunden werden, ist der Drucker auszuschalten.

Durch die Option ***"Dateiname"*** kann die Ausgabe auch in die angegebene Datei umgelenkt werden (s. **LIST**).

LPOS() — Druckkopfposition ermitteln

```
Var=LPOS(Dummy)
```



LPOS() ermittelt die Position des virtuellen Druckkopf innerhalb des Drucker-Puffers. ***'Dummy'*** ist ein beliebiges Scheinargument ohne Bedeutung.

LPRINT { LPR } — Daten auf Drucker ausgeben

```
LPRINT [,'] ''Text'' [[;,'] Var [;,'] Expr...]
```



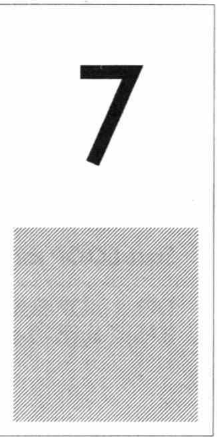
Es gelten weitestgehend diesselben Ausführungen wie zu **PRINT**. Eine Druckkopf-Positionierung mit Hilfe des Zusatzes ***AT*** ist allerdings nicht möglich.

Zu den *Steuersequenzen* Ihres Druckers (*Escapes*) beachten Sie bitte Ihr Druckerhandbuch.

Notizen:

Strukturen und Verzweigungen

7



7. STRUKTUREN UND VERZWEIGUNGEN

7.1. SCHLEIFEN - KONSTRUKTIONEN

DO ... LOOP { DO ... L } Endlosschleife

```
DO
 ...  auszuführende Programmteile
LOOP

oder:

DO [WHILE Bedingung] [UNTIL Bedingung]

  ...  auszuführende Programmteile, wenn die
  ...  DO-'Bedingung' wahr ist, bzw.
  ...  solange die LOOP-'Bedingung' wahr ist.
  .
LOOP [WHILE Bedingung] [UNTIL Bedingung]
```

Eine *'normale'* DO...LOOP-Schleife kann nur abgebrochen werden, wenn sie auf eine Abbruch-Anweisung (**END, EDIT, STOP**) trifft, eine **EXIT IF**-Anweisung findet und die Abbruchbedingung wahr ist, eine **GOTO**-Anweisung (bitte nicht!!) innerhalb der Schleife zu einem Label außerhalb der Schleife verzweigt oder die Break-Funktion (s. **ON BREAK**...) verwendet wird.

Mit der zweiten Syntax-Variante ist es möglich, eine DO...LOOP-Konstruktion mit Eingangs- u n d Ausgangsbedingungen zu versehen. Dazu kann sowohl bei DO, als auch bei LOOP entweder eine WHILE- oder eine UNTIL-Bedingungsabfrage hinzugefügt werden (s. **WHILE...WEND**, bzw. **REPEAT...UNTIL**). Allgemein ist es möglich, fast alle erdenklichen Kombinationen von WHILE...WEND-, REPEAT...UNTIL- und DO...LOOP-Schleifen zu realisieren.

Statt ***LOOP*** kann synonym auch ***ENDDO*** verwendet werden.

DO...LOOP-Schleifen können - wie alle anderen Schleifen in GFA-BASIC auch - beliebig tief verschachtelt werden.

FOR ... NEXT { F ... N } Zählschleife

```
FOR Zaehl=Anf TO [DOWNTO] Ende [STEP Schritt]
   ...  auszuführende Programmteile
NEXT Zaehl
```

Die Kopfzeile der Schleife enthält den Anfangswert ***'Anf'*** und den Endwert ***'Ende'***. Die Zählvariable ***'Zaehl'*** wird, beginnend mit ***'Anf'*** solange erhöht, bzw. vermindert, bis sie den Wert ***'Ende'*** erreicht hat. Danach wird das Programm mit der auf die NEXT-Anweisung folgenden Programmzeile fortgesetzt. Wird nur FOR TO / NEXT ohne die Zusätze ***DOWNTO*** oder ***STEP*** verwendet, beträgt die Schrittweite immer +1 (s. Beispiel zu ***XLATE$***).

Bei Verwendung von DOWNTO statt TO ist der Anfangswert größer als der Endwert anzugeben, da in diesem Fall die Schrittweite immer -1 beträgt. Die Option ***STEP*** (nur bei ***TO***-Schleifen) bewirkt, daß der nach ***STEP*** angegebene Wert oder Ausdruck als Schrittweite angenommen wird. Hier sind auch negative Werte möglich.

Statt ***'NEXT Var'*** kann auch ***'ENDFOR Var'*** **{ ENDFO }** angegeben werden.

REPEAT ... UNTIL { REP ... U } End-bedingte Schleife

```
REPEAT
...  auszuführende Programmteile
UNTIL Bedingung
```

Die Bedingung zum Schleifen-Exit wird am Schleifenende geprüft. D.h., daß das Programm die Schleife mindestens einmal bis zu der in UNTIL vereinbarten Bedingung durchläuft (falls keine zusätzlich in der Schleife angeordnete **EXIT IF**-Bedingung mit *´wahr´* beantwortet wird). Ist ***'Bedingung'*** wahr, wird das Programm mit der nächsten auf UNTIL folgenden Zeile fortgesetzt (s. Beispiel zu **XLATE$**).

Statt ***'UNTIL Bedingung'*** kann auch ***'ENDREPEAT Bedingung'*** **{ ENDR }** angegeben werden.

WHILE ... WEND { W ... WE } Start-bedingte Schleife

```
WHILE Bedingung
...   auszuführende Programmteile
WEND
```

Die Exit-Bedingung wird am Schleifenanfang geprüft. D.h., daß die Schleife nicht durchlaufen wird, falls die bei WHILE vereinbarte ***'Bedingung'*** wahr ist. Ist ***'Bedingung'*** wahr, wird das Programm mit der nächsten auf WEND folgenden Zeile fortgesetzt.

Statt ***WEND*** kann auch ***ENDWHILE*** **{ ENDW }** angegeben werden.

7.2. BEDINGTE VERZWEIGUNGEN

EXIT IF { EX IF } Bedingter Schleifenabbruch

```
EXIT IF Bedingung
```

Es kann unabhängig vom Zustand einer Schleife diese jederzeit verlassen werden, wenn die durch EXIT IF gestellte ***'Bedingung'*** wahr ist. Das Programm wird dann mit der nächsten Programmzeile hinter dem nächsten Schleifenwendepunkt (**LOOP, NEXT, WEND** etc.) fortgesetzt. In allen Arten von Schleifen sind an beliebigen Stellen beliebig viele EXIT IF's möglich.

IF — Bedingungsabfrage

...

[ELSE] {EL} — 'sonst'-Anweisung

...

[ELSE IF] {E IF} — Unter-Bedingungsabfrage

...

ENDIF {EN} — Abfrage-Ende

```
IF Bedingung [THEN]
  ...  auszuführende Programmteile,
  ...  wenn 'Bedingung' wahr ist
[ELSE
  ...  auszuführende Programmteile,
  ...  wenn 'Bedingung' unwahr ist ]
ENDIF

oder:

IF Bedingung1 [THEN]
  ...  auszuführende Programmteile,
  ...  wenn 'Bedingung' wahr ist
[ELSE IF Bedingung2
  ...  auszuführende Programmteile,
  ...  wenn 'Bedingung1' unwahr
  ...  und 'Bedingung2' wahr ist.]
[ELSE IF Bedingung3
  ...  auszuführende Programmteile, wenn
  ...  alle voherigen Bedingungen unwahr
  ...  waren, jedoch 'Bedingung3' wahr ist.]
[gfls. weitere ELSE IF-Abfragen]
  ...
[ELSE
  ...  auszuführende Programmteile,
  ...  wenn alle vorherigen Bedingungen
  ...  unwahr waren.]
ENDIF
```

Ist ***'Bedingung'*** wahr, werden die zwischen IF und ENDIF stehenden Programmteile ausgeführt. Bei Verwendung der Option ***ELSE*** werden die darauffolgenden Programmteile ausgeführt, wenn ***'Bedingung'*** unwahr ist.

IF-Abfragen können beliebig oft verschachtelt werden. Der optionale Zusatz ***THEN*** hinter IF ist nur zur Kompatibilität mit anderen BASIC-Dialekten gedacht. Er kann vernachlässigt werden.

Durch den Zusatzbefehl ***ELSE IF*** ist es möglich, Verschachtelungen folgender Art zu ersparen:

```
IF Bedingung1
  ...
ELSE
  IF Bedingung2
    ...
    gfls. weitere Verschachtelungen
    ...
  ELSE
    ...
  ENDIF
ENDIF
```

Mit ***ELSE IF*** sieht dieselbe Struktur so aus:

```
IF Bedingung1
  ...
ELSE IF Bedingung2
  ...
gfls. weitere ELSE IF's
  ...
ELSE
  ...
ENDIF
```

Ist die Eingangs-IF-Bedingung unwahr und trifft das Programm auf eine ELSE IF-Abfrage, deren Bedingung wahr ist, wird nur der darunter angegebene Programmblock abgearbeitet und nach dessen Ausführung zu dem nächsten Befehl hinter der zugehörigen ENDIF-Anweisung gesprungen. Wird die Option ELSE verwendet und keine der vorangegangenen Bedingungen war wahr, wird die unter ELSE angegebene Programmfolge ausgeführt.

SELECT		Auswahl-Bestimmung
CASE [TO]	**{ CA }**	Fall-Entscheidung
...		
[CONT]	**{ CON }**	Fortsetzungs-Anweisung
[DEFAULT]	**{ DEFA }**	'sonst'-Anweisung
...		
ENDSELECT	**{ ENDS }**	Abfrage-Ende

```
SELECT Expr (oder SWITCH Expr)
CASE Constant1 [TO Constant2 [,[..] TO [..]]]
  ... auszuführende Programmteile, wenn 'Expr'
  ... gleich Constant1, bzw. - bei Option TO -
  ... wenn 'Expr' innerhalb des Bereichs von
  ... Constant1 bis Constant2 liegt.
  [CONT]
[CASE Constant1 [,Constant2 [,Constant3 [,..]]]
  ... auszuführende Programmteile, wenn 'Expr'
  ... gleich Constant1  o d e r  gleich Constant2
  ... o d e r  gleich Constant3  o d e r ...
  ... o d e r ... o d e r ...]
[gfls. weitere CASE-Entscheidungen]
  ...
  [CONT]
[DEFAULT oder OTHERWISE oder CASE ELSE
  ... auszuführende Programmteile, wenn keine der
  ... vorhergehenden Abfragen zugetroffen hat.]
ENDSELECT oder ENDSWITCH
```

Diese Form der Bedingungsabfrage bietet die Möglichkeit zur ***'Expr'***-abhängigen Programmverzweigung (in *'C': Switch/Case*). ***'Expr'*** kann ein beliebiger numerischer oder alphanumerischer Ausdruck sein, dessen Ergebnis gfls. vorher ermittelt wird. Es ist auch die Angabe von Variablen oder Konstanten möglich. Wird ein alphanumerischer Ausdruck, eine Text-Konstante oder -Variable in ***'Expr'*** verwendet, werden davon nur die ersten vier Zeichen zum Vergleich herangezogen. Diese werden dann intern in einen 4Byte-Wert umgewandelt.

Hinter CASE wird bei Werte-SELECT in ***'Constant..'*** ein numerischer Wert oder ein max. vier Zeichen langer Text als Konstante oder Stringvariable angegeben, der dann daraufhin überprüft wird, ob ***'Expr'*** ihm entspricht. Bei Werte-SELECT angegebene Strings

werden auf Gültigkeit geprüft, indem der SELECT-Wert mit den ASCII-Werten der ersten vier Zeichen des Textes (sofern vorhanden) verglichen wird.

z.B.:

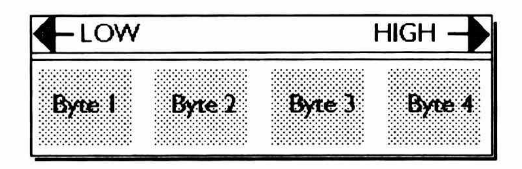
```
c$=SPACE$(4)          //  4Byte-Kontrollstring
FOR i%=0 TO 3         //  4 mal
  PRINT CHR$(10);''Bitte ASCII-Taste drücken:''
  KEYGET a|           //  Einzelzeichen holen
  b%=b%+a|*2^(i%*8) //  Zeichen einsortieren
  {V:c$}={V:b%}       //  Kontrollstring kopieren
  PRINT BIN$(b%)''c$ //   zur Kontrolle ausgeben
NEXT i%               //  nächstes Zeichen
SELECT b%             //  eingegebenes Long testen
CASE ''abcd''         //  'abcd' eingegeben?
  PRINT CHR$(10);''abcd wurde eingegeben''
DEFAULT               //  irgendwas eingegeben!
  PRINT CHR$(10);MKL$(b%)'''wurde eingegeben!''
ENDSELECT             //  Abfrage-Ende
```

Es wird also intern ein **MKL$-, MKI$-** oder **CHR$**-String gebildet (je nach Länge des angegebenen CASE-Strings), der dann mit ***'Expr'*** verglichen wird. Hierbei ist das MSDOS-Datenformat zu beachten: der ASCII-Wert des ersten Zeichens des Strings (von links ausgehend) liegt im Speicher vorn, also bei einem 4Byte-Integer auch *'links'* im LO-Byte des LO-Words.

z.B. bei einem 4-Zeichenstring:

  (ASCII-Wert des 1. Zeichens)
+ (ASCII-Wert des 2. Zeichens) * (2^8)
+ (ASCII-Wert des 3. Zeichens) * (2^16)
+ (ASCII-Wert des 4. Zeichens) * (2^24)

ergibt im Speicher:



etc.

Bei String-SELECT ist ebenfalls nur die Angabe eines maximal vier Zeichen langen Strings hinter CASE zulässig.

Entspricht ***'Expr'*** der CASE-Auswahl, wird die darauffolgende Programmsequenz ausgeführt und daran anschließend zu der nächsten Zeile hinter dem zugehörigen ENDSELECT gesprungen.

Evtl. weitere **CASE**-Abfragen derselben Gruppe bleiben dann also unberücksichtigt.

Durch die optionale Angabe von ***TO*** kann ein ganzer Bereich angegeben werden

z.B.

```
CASE 1 TO 10
CASE ''a'' TO ''z''
```

oder

```
CASE ''abc'' TO ''xyz'')
```

innerhalb dessen Grenzen ***'Expr'*** liegen muß, um die zugehörige Sequenz zu durchlaufen.Wird die erste (kleinere) Bereichsgrenze vor ***TO*** oder die zweite (größere) Bereichsgrenze nach ***TO*** weggelassen, wird intern automatisch die kleinstmögliche bzw. größtmögliche Grenze angenommen.

Durch Verwendung eines Kommas als Trennzeichen können auch mehrere Einzelangaben zusammengefaßt werden

z.B.

```
CASE a,h,j,m
```

oder

```
CASE 1,33,7)
```

Es ist möglich, die **CASE**-Bedingungsformate in einer **CASE**-Zeile beliebig zu vermischen

z.B.

```
CASE  TO ''b'',''ABC'' TO ''XYZ'',65,66,67,''Ä'')
```

Wurden sämtliche angegebenen **CASE**-Anweisungen ohne *'wahr'*-Ergebnis passiert, kann am Ende des **SELECT**-Blocks **DEFAULT** eingesetzt werden, was dazu führt, daß dann der zwischen **DEFAULT** und **ENDSELECT** liegende Programmteil ausgeführt wird (vergleichbar mit **ELSE** bei **IF**-Abfragen).

Wird direkt vor einer **CASE**-Anweisung am Ende eines Verzweigungsblocks die Option **CONT** verwendet, bewirkt dies, daß die direkt danach stehende **CASE**-Abfrage übersprungen wird und die dieser **CASE**-Abfrage unterstellte Sequenz zusätzlich zur schon ausgeführtenVerzweigung ebenfalls ausgeführt wird. Nach Erledigung dieser Folgesequenz wird - sofern nicht auch diese mit **CONT** abgeschlossen wurde - zur ersten Zeile hinter **ENDSELECT** gesprungen. Dasselbe ist auch mit **DEFAULT** möglich. Steht **CONT** nicht direkt vor **CASE** oder **DEFAULT**, wird es als **CONT** zur Programmfortsetzung nach einem **STOP**-Befehl interpretiert.

Statt **SELECT** kann auch **SWITCH { SWI }**, statt **DEFAULT** auch **OTHERWISE { OT }**, oder **CASE ELSE { C ELSE}** und statt **ENDSELECT** auch **ENDSWITCH { ENDSW }** angegeben werden.

SELECT...CASE- und **IF...ENDIF**-Konstruktionen können beliebig - auch beliebig tief - miteinander verschachtelt werden. Wo immer möglich, sollten aus Geschwindigkeitsgründen **IF...ENDIF**-Konstruktionen durch **SELECT...CASE**'s ersetzt werden.

7.3. UNTERPROGRAMME UND STRUKTUREN

DEFFN einzeilige Funktion definieren

```
DEFFN Name[(Var,[Var2,...])]=Expr
```

'Name' steht für einen beliebigen Funktionsnamen, durch welchen die Funktion mittels des Funktionsaufrufs **FN** angesprochen werden kann. Zur Namensbildung können bereits bestehende Variablennamen verwendet werden. Das erste Zeichen kann, anders als bei Variablen, auch eine Ziffer sein.

Mit dem optionalen Zusatz ***'Var,Var2...'*** können mehrere, durch Komma getrennte Variablen angegeben werden, denen ein evtl. mit **FN** übergebener Wert zugeordnet wird. Diese Liste kann auch Variablen verschiedener Typen enthalten, solange die mit **FN** zu übergebenden Daten in ihrer Reihenfolge zum gleichen Typ gehören.

Innerhalb der DEFFN-Funktion sind nur Operationen erlaubt, die dem verwendeten Funktionstypen entsprechen. D.h., wenn die Funktion als Stringtyp deklariert wurde (z.B. DEFFN Funktion$=...) sind nur Stringoperationen möglich. Beim Aufruf der Funktion werden die z.Zt. aktuellen Inhalte der darin verwendeten Variablen angenommen.

Die Länge einer DEFFN-Funktion wird durch die maximale Eingabezeilenlänge im GFA-BASIC-Editor (256 Zeichen) beschränkt. Reicht diese Länge nicht aus, können aus einer Funktion heraus andere Funktionen aufgerufen werden. DEFFN-Funktionen können an jeder beliebigen Stelle des Programms definiert werden, da schon bei Programmstart alle DEFFN-Funktionen initialisiert werden.

Vorsicht:
Endlosschleifen, worin sich zwei Funktionen gegenseitig aufrufen, können auch durch die Break-Funktion (s. ON BREAK...) nicht mehr unterbrochen werden. Funktionen, die sich ohne Abbruchbedingung rekursiv selbst aufrufen, haben denselben Effekt.

FUNCTION { **FU** } mehrzeilige Funktion

...

RETURN { **RET** } Wertrückgabe-Anweisung

ENDFUNC {**ENDF** } Funktionsende

```
FUNCTION Name [(Var1,Var2%,Var3$,...)]
  ... auszuführende Programmteile
  RETURN Back
ENDFUNC
```

FUNCTION kennzeichnet den Anfang einer selbstdefinierten, mehrzeiligen Funktion. Es kann optional eine Liste von lokalen Variablen (s. **LOCAL**) angegeben werden, welche die durch den Funktionsaufruf FN übergebenen Daten aufzunehmen haben.

Dabei ist darauf zu achten, daß die Variablen in dieser Liste der Reihenfolge nach den Datentypen der gfls. durch **FN** übergebenen Parametern entsprechen. Es ist auch möglich, durch **VAR** als Abschluß dieser Liste Variablen zu definieren, an die dann durch **FN** eine globale Variable direkt übergeben werden kann (s. **VAR**).

'Name' ist ein beliebiger Name, der die Funktion benennt. Soll die Funktion alphanumerische (String-) Ergebnisse liefern, ist dem Funktionsnamen ein '**$**' anzuhängen

z.B.

```
FUNCTION Name$(...)).
```

Innerhalb der **FUNCTION** kann - wie bei einer **PROCEDURE** - beliebig viel Programmtext angeordnet werden. Es ist auch möglich, FUNCTION-Funktionen sich selbst aufrufen zu lassen (*Rekursion*).

ENDFUNC bildet den strukturellen Abschluß einer FUNCTION. Im Gegensatz zum **PROCEDURE-RETURN** wird diese Endmarkierung nicht als Rücksprunganweisung interpretiert. Aus FUNCTION-Funktionen ist nur ein Rücksprung durch ***'RETURN Back'*** möglich, was nicht mit dem Prozedurende **RETURN** verwechseln werden darf. Trifft das Programm innerhalb einer

FUNCTION auf die Rücksprunganweisung RETURN, wird der hinter RETURN stehende Wert, Ausdruck oder Variableninhalt ***'Back'*** als Funktionsergebnis an das aufrufende Programm zurückgegeben und zu der dem Aufruf folgenden Zeile gesprungen, bzw. - falls die Funktion aus einer Befehlszeile heraus aufgerufen wurde - mit der Befehlsausführung hinter dem Funktionsaufruf fortgefahren. Innerhalb der Funktion geänderte globale Variablen oder Felder können in diesem Fall gfls. schon bei der Abarbeitung des Zeilenrestes berücksichtigt werden.

Es können beliebig viele ***'RETURN Back'***-Anweisungen innerhalb einer FUNCTION angegeben werden (z.B. innerhalb von **IF..ELSE IF..ELSE..ENDIF**-Abfragen).

z.B.:

```
SCREEN 16
COLOR 14
FOR i%=0 TO 72
  POLYLINE LEN(@cbox$(300,180,140,100,...
               ...360-i%*6,3,3))/4,px%(),py%()
  POLYLINE LEN(@cbox$(300,180,220,160,...
               ...i%*4+240,5,2))/4,px%(),py%()
NEXT i%
'
FUNCTION cbox$(xp%,yp%,xr%,yr%,wi%,e%,m%)
  '
  ' Berechnet ein beliebiges gleichmäßiges
  ' Vieleck und liefert die Eck-Koordinaten
  ' in einem Integer-Feld zurück
  '
  ' xp% = X-Mittelpunkt
  ' yp% = Y-Mittelpunkt
  ' xr% = X-Radius
  ' yr% = Y-Radius
  ' wi% = Startwinkel
  ' e%  = zu berechnende Eckenanzahl
  ' m%  = Anzahl zu zeichnender Ecken
  '
  LOCAL j%,i%,bk$
  ERASE px%(),py%()
  DIM px%(e%),py%(e%)
  FOR i%=wi% TO 360+wi% STEP 360/e%
    px%(j%)=xp%+(SINQ(i%)*xr%)
    py%(j%)=yp%+(COSQ(i%)*yr%)
    bk$=bk$+MKI$(px%(j%))+MKI$(py%(j%))
    IF j%=>m% THEN RETURN bk$
    j%++
  NEXT i%
ENDFUNC
```

PROCEDURE { PRO } Unterprogramm

••

RETURN { RET } Rücksprung

```
PROCEDURE Name [([Var1,Var2%,Var3$,VAR Var4,...])]
  ... auszuführende Programmteile
RETURN
```

PROCEDURE definiert den Anfang eines Unterprogramms. Es kann optional eine Liste von lokalen Variablen (s. **LOCAL**) angegeben werden, welche die durch den Prozeduraufruf **GOSUB** übergebenen Daten aufzunehmen haben. Dabei ist darauf zu achten, daß die Variablen in dieser Liste der Reihenfolge nach den Datentypen der gfls. durch **GOSUB** übergebenen Parametern entsprechen. Es ist auch möglich, durch **VAR** als Abschluß dieser Liste Variablen zu definieren, an die dann durch **GOSUB** eine globale Variable direkt übergeben werden kann (s. **VAR**).

'Name' ist ein beliebiger Name, der das Unterprogramm benennt. Innerhalb der Routine kann beliebig viel Programmtext angeordnet werden. Es ist auch möglich, Prozeduren sich selbst aufrufen zu lassen (*Rekursion*).

RETURN bildet den Abschluß einer Prozedur und bewirkt eine Fortsetzung des Programms mit der auf den zugehörigen **GOSUB**-Befehl folgenden Zeile.

z.B.:

```
SCREEN 16
COLOR 14
FOR i%=0 TO 90 STEP 3
  cbox(200,80,i%*2,i%,i%,6,vec$)
  POLYLINE LEN(vec$)/4,px%(),py%()
NEXT i%
cbox(520,130,50,30,0,12,vec$)
FOR i%=0 TO LEN(vec$)-1 STEP 4
  PCIRCLE INT{V:vec$+i%},INT{V:vec$+i%+2},6
NEXT i%
'
PROCEDURE cbox(xp%,yp%,xr%,yr%,wi%,e%,VAR bk$)
  '
  ' Berechnet ein beliebiges gleichmäßiges
  ' Vieleck und liefert die Eck-Koordinaten
  ' als MKI$-Vector in einem Textstring zurück
  '
  ' xp%/yp%/xr%/yr%/wi%/e% = s. unter FUNCTION
  '
  ' bk$ = Stringvariable, die bei Rückkehr den
  '       MKI$-Koordinaten-String aufnimmt.
  '
```



```
    LOCAL j%,i%
    CLR bk$
    ERASE px%(),py%()
    DIM px%(e%),py%(e%)
    FOR i%=wi% TO 360+wi% STEP 360/e%
      px%(j%)=xp%+(SINQ(i%)*xr%)
      py%(j%)=yp%+(COSQ(i%)*yr%)
      bk$=bk$+MKI$(px%(j%))+MKI$(py%(j%))
      j%++
    NEXT i%
  RETURN
```

TYPE { **TY** } — Typenstruktur definieren

...

ENDTYPE { **ENDT** } — Typenstruktur-Ende

```
TYPE Typenname:
- ELEMENTE-TYP-1  Elementname1
- ELEMENTE-TYP-2  Elementname2
- etc.
ENDTYPE
[Typenname:Variablenname.]
```

TYPEs sind überaus wichtige Strukturelemente, (in PASCAL: *'records'*, in C: *'structures'*), die es ermöglichen, komplexe Datenstrukturen nach den verschiedensten Notwendigkeiten selbst zu erstellen. Die zusammengehörigen Elemente eines so erstellten Datentyps sind dann einerseits leicht unter ihrem jeweiligen ***'Variablennamen.'*** und ***'Elementnamen'*** ansprechbar und können andererseits jederzeit für die vielfältigsten Aufgaben als einheitlicher Datensatz behandelt werden. So kann z.B. ein solcher Strukturblock anhand eines einzigen **BMOVE**- oder **BSAVE**-Befehls an beliebige Positionen im RAM oder auf dem Festspeicher verfrachtet werden. Dabei ist es wichtig, zu verstehen, daß die Declaration einer TYPE-Struktur selbst noch nichts über die Lage dieser Struktur im Speicher aussagt.

Wer sich schon mit dem Einsatz von *'Random-Access-Dateien'* und dem **FIELD**-Befehl vertraut gemacht hat, wird den Sinn einer TYPE-Struktur sicher leicht nachvollziehen können. Auch in der objektorientierten Programmierung sind Datenstrukturen - z.B. für *'DTP'*-, *'Window'*- oder auch *'CAD/CAM'*-Objekte - Gang und Gäbe.

Der ***'Typname:'*** einer TYPE-Struktur kann in beliebiger Form (wie ein **PROCEDURE**- oder **FUNCTION**-Name) vergeben werden. Er endet zur internen Kennzeichnung einer TYPE-Struktur mit einem Doppelpunkt.

Vor dem ***'ELEMENTE-TYP'*** ist ein Bindestrich und ein darauffolgendes Leerzeichen zu placieren. Hinter dem ***'ELEMENTE-TYP'*** folgt mit dem Abstand von mindestens einem Leerzeichen der dazugehörige ***'Elementname'***. Dieser ***'Elementname'*** gehört ausschließlich zu *'seiner'* TYPE-Struktur und kann nur einmal vergeben werden.

Zur Definiton der einzelnen *'ELEMENTE-TYPEN'* stehen folgende Datenformate zur Verfügung:

Art:	*Format:*
- BYTE	8Bit-Byteformat
	vorzeichenlos (Kennung |)
- UBYTE	0 bis 255
- CARD	16Bit-Wortformat
- UWORD	vorzeichenlos (unsigned)
- USHORT	0 bis 65535
- WORD	16Bit-Wortformat
	vorzeichenbehaftet (Kennung &)
- SHORT	-32768 bis 32767
- INT	32Bit-Langwortformat
	vorzeichenbehaftet (Kennung %)
- LONG	-2147483648 bis 2147483647
- SINGLE	4Byte (einfach genau)
	IEEE-Fließkommaformat (Kennung #)
- DOUBLE	8Byte (doppelt genau)
- CHAR(n)	beliebige Zeichenkette
- CHAR*n	mit einer maximalen Länge
- STRING(n)	von
- STRING*n	*'n'* Zeichen (Kennung $)

Bei Verwendung von TYPE-Variablen der Art ***'CHAR'*** oder ***'STRING'***, bzw. der Art ***'SINGLE'*** und ***'DOUBLE'*** muß die Typenkennung ***'$'*** bzw. ***'#'*** dem Variablennamen angefügt werden.

Sollen die Struktur-Elemente im späteren Programm unter ihren ***'Elementnamen'*** wie normale Variablen als intern verwaltete Variablen direkt angesprochen werden können, muß vorher ein TYPE-***'Variablenname'*** bestimmt werden. Dazu wird im Anschluß an die TYPE-Definition hinter ***'Typname:'*** der dazugehörige ***'Variablenname.'*** deklariert. Er erhält als Kennung einen abschließenden Punkt.

z.B.: `typname:variablenname.`
oder `DIM typname:var1.,typname:var2.,...`

Von nun an existiert faktisch ein Speicherbereich mit der spezifischen TYPE-Länge, der unter dem angegebenen ***'variablennamen.'*** *(Punktendung beachten!)*, bzw. unter dessen Variablen-***'elementnamen'*** ansprechbar ist.

z.B.: `variablenname.elementname$=''Text''`
oder `txt$=variablenname.elementname$`
oder `PRINT variablenname.elementname$`

Aus der Verwendung des Punktes als Kennzeichnung für TYPE-Variablen bzw. die Elemente ergibt sich logisch, daß die Verwendung eines Punktes innerhalb normaler Variablennamen nicht zulässig ist.

Statt des Struktur-Zugriffs per ***'Variablenname.'*** kann auch ein Zeiger auf den Anfang einer entsprechenden Struktur mit dem nachfolgenden Punkt und dem wiederum darauf folgenden ***'Elementnamen'*** angegeben werden.

z.B.: `{adr}.element=expr`
oder `var={adr}.element`

Um die spätere Ansprechbarkeit der entsprechenden TYPE-Elemente über ihren ***'Elementnamen'*** im Programm möglichst einfach zu gestalten, sollte darauf geachtet werden, daß ***'Variablenname.'*** und ***'Elementname'*** kurz gehalten werden. Bei häufiger Verwendung von TYPEs erspart man sich so viel Tipperei.

Auch die Einrichtung von TYPE-Feldern ist möglich, indem man ein Text-Feld dimensioniert, zu Beginn einmal über eine Init-Schleife mit den korrespondierenden Elementlängen vorbereitet und die zusammengehörigen TYPE-Blöcke in einem solchen Feld unterbringt. Das einzige Problem besteht dann darin, daß die Startadresse des Textfeldes aufgrund der dynamischen Stringspeicherverwaltung evtl. nicht konstant bleibt und vor Direkt-Zugriffen auf eine TYPE (z.B. `{adr}.var`) eine *'Carbage Collection'* mit **'FRE(0)'** ausgeführt werden muß. Anschließend kann dann durch **VARPTR()** die Startadresse des ersten Strukturelements und damit die Startadresse der gesamten TYPE ermittelt werden. Eine weitere Möglichkeit besteht darin, die Startadresse des Textfeld-Elementes direkt anzugeben (z.B. `{V:textfeld$(0)}.var`)

z.B.

```
TYPE irgendwas:         // Beispiel-Deklaration
- CHAR*20 txt$          // z.B. 20 Zeichen Text
- LONG    l1            //     z.B.
- LONG    l2            //   2 Longs
- CARD    crd           // z.B. 1 UWORD
ENDTYPE                 // TYPE-Ende
DIM feld$(20)           // z.B. 21 Elemente
FOR i%=0 TO 20          // alle Elemente
  feld$(i%)=STRING$(LEN(irgendwas:),0)//'nullen'
NEXT i%
~FRE(0)                 // Garbage-Collection
{V:feld$(0)}.l1=12345 // irgendeine Zuweisung
{V:feld$(1)}.l1={V:feld$(0)}.l1^2  // irgendeine
wert&=AND({V:feld$(0)}.l1+{V:feld$(1)}.l1,$FFFF)// Be-
{V:feld$(1)}.crd=wert&             // rechnung
FOR i%=0 TO 1               // z.B. die ersten zwei
  PRINT {V:feld$(i%)}.l1  // Elemente ausgeben
NEXT i%
PRINT {V:feld$(1)}.crd     // UWORD aus Element '1'
OPEN "O",#1,"MEINTYPE.REC"// z.B. komlettes
FOR i%=0 TO 20              // TYPE-Feld in Datei
  BPUT #1,V:feld$(i%),len(irgendwas:) // ablegen
NEXT i%
CLOSE #1
```

Um sich nicht von der dynamischen Speicherverwaltung ärgern lassen zu müssen, kann man statt eines Textfeldes natürlich auch ein entsprechend langes Integerfeld verwenden, das dann statisch behandelt werden kann. Nur hat man hier wiederum die Problematik der korrekten Indizierung, was aber mit etwas *'KnoffHoff'* ohne weiteres zu bewältigen ist:

```
TYPE irgendwas:         // Beispiel-Deklaration
- CHAR*20 txt$          // z.B. 20 Zeichen Text
- LONG    l1            //     z.B.
- LONG    l2            //   2 Longs
- CARD    crd           // z.B. 1 UWORD
ENDTYPE                 // TYPE-Ende
slen%=LEN(irgendwas:) // TYPE-Länge ermitteln
DIM feld%((slen%*21)/4+4) // INT-Feld mit
'                       // Gesamtlänge dimmen (geteilt
'                       // durch vier, weil ein INT
'                       // 4 Bytes lang ist)
ARRAYFILL feld%(),0     // ARRAYFILL ist hier eigentlich
'                       // nicht nötig, da nach einem DIM
'                       // sowieso alle Elemente Null sind
adr%=V:feld%(0)         // Startadresse des Puffers
{adr%+0*slen%}.l1=12345 // z.B. irgendeine Zuweisung
'                       // an 'l1' im Element 0
{adr%+1*slen%}.l1={adr%+0*slen%}.l1^2 // z.B. irgendeine
'                       // Zuweisung an 'l1' im Element 1
wert&=AND({adr%+0*slen%}.l1+{adr%+1*slen%}.l1,$FFFF)
'                       // z.B. irgendweine Berechnung
'                       // mit 'l1' aus Element 0 und 1
{adr%+1*slen%}.crd=wert& // Zuweisung an 'crd' in Element 1
FOR i%=0 TO 1             // z.B. die ersten zwei
```

```
  PRINT {adr%+i%*slen%}.l1 // Elemente ausgeben
NEXT i%
PRINT {adr%+1*slen%}.crd    // z.B. 'crd' aus Element 1
'                           // ausgeben
BSAVE ''MEINTYPE.REC'',adr%,slen% // z.B. komlettes TYPE-
'                           // Feld in Datei ablegen
```

Es ist auch möglich, TYPE-Variablen untereinander zuzuweisen. Dazu müssen die einzelnen Variablen allerdings dieselbe Struktur aufweisen.

z.B.

```
TYPE irgendwas:         // Beispiel-Deklaration
- CHAR(6) txt$          // z.B. 6 Zeichen Text
- LONG    lvar          // z.B. 1 Long
- BYTE    bvar          // z.B. 1 Byte
ENDTYPE                 // TYPE-Ende
irgendwas:variable1.
irgendwas:variable2.
variable1.txt$=''*TEXT*''
variable1.lvar=123456
variable1.bvar=255
variable2.=variable1.
```

Die Adresszeiger-Funktionen des GFA-BASIC sind auch auf TYPE-Variablen anwendbar:

```
adr%=VARPTR(typ_var.)
adr%=V:typ_var.
adr%=ARRPTR(typ_var.)
adr%=*typ_var.
```

und auch direkte Speicherzugriffe im TYPE-Format sind möglich:

```
TYPE irgendwas:         // Beispiel-Deklaration
- CHAR*20 txt$          // z.B. 20 Zeichen Text
- LONG    l1            //     z.B.
- LONG    l2            //   2 Longs
- CARD    crd           // z.B. 1 UWORD
ENDTYPE                 // TYPE-Ende
irgendwas:typ_var.
slen%=LEN(irgendwas:) // TYPE-Länge ermitteln
DIM feld%((slen%*21)/4+4) // INT-Feld dimmen
adr%=V:feld%(0)         // Startadresse des Puffers
'
... Programm
'
' z.B.:
typ_var.=irgendwas:{V:feld%(0)}
' TYPE-PEEK vom Speicher in eine TYPE-Variable
' derselben Struktur
' -> entspricht: var%=LPEEK(adr%)
' z.B.:
irgendwas:{V:feld%(0)}=typ_var.
' TYPE-POKE aus einer TYPE-Variablen in einen
' Speicherbereich mit derselben Struktur
 ' -> entspricht: LPOKE adr%,var%
```




```
' z.B.:
irgendwas:{V:feld%(1)}=irgendwas:{V:feld%(0)}
' TYPE-MOVE aus einem Speicherbereich in einen
' anderen Bereich mit derselben Struktur
' -> entspricht: BMOVE adr1%,adr2%,slen%
```

LOCAL { LOC } Lokale Variablen deklarieren

```
LOCAL var [,var2%,var3$,...]
LOCAL var=wert [,var2%=wert,var3$=''Text'',...]
```

Es können innerhalb einer **PROCEDURE** oder **FUNCTION** Variablen als lokale Variablen deklariert werden (s. Beispiel zu **FUNCTION**). Diese können ausschließlich in der **PROCEDURE** oder **FUNCTION** verwendet werden, in welcher sie deklariert wurden. Nach Prozedur- bzw. Funktionsende werden sie wieder aus dem BASIC-Gedächtnis gelöscht.

Wird außerhalb dieser Prozedur/Funktion eine globale Variable mit gleichem Namen benannt, so ist GFA-BASIC in der Lage, diese von denen durch LOCAL deklarierten Variablen zu unterscheiden. Werden mehrere Variablen deklariert, können sie unterschiedlichen Typs sein und sind durch Kommata zu trennen.

Es ist auch möglich, direkt hinter der LOKAL-Deklaration eine bzw. mehrere Zuweisung(en) zu placieren.

7.4. SPRÜNGE UND VARIABLENÜBERGABE

FN {@} DEFFN-/FUNCTION-Aufruf

```
Var=FN Funktionsname [(Para1,Para2%,Para3$,...)]
```

Wurde bei **DEFFN** oder bei **FUNCTION** eine Variablen-Liste vorgegeben, müssen beim Funktionsaufruf in ***'Para1, Para2%, Para3$, ...'*** - durch Kommata getrennt - ebenso viele und dem jeweiligen Variablentyp entsprechende Werte, Strings etc. übergeben werden (s. Beispiel zu **FUNCTION**).

Die Ergebnisse einer Funktion können entweder direkt ausgegeben (z.B. `PRINT @Funk`), einer dem Funktionstyp entsprechenden Variablen übergeben (`A%=@Funk`), in Bedingungsabfragen

ausgewertet (z.B. `IF @Funk...`), in arithmetische Konstrukte oder Text-Ausdrücke eingebunden

z.B. `DIV Var1,(Expr+Var2)/(Constant*@Funk)`

oder ignoriert werden (z.B. `VOID @Funk`).

GOSUB { GO oder @ } PROCEDURE-Aufruf

```
GOSUB Prozedurname [(Para1,Para2%,Para3$,...)]
```

'Prozedurname' gibt den Namen der **PROCEDURE** an, zu der verzweigt werden soll. Es können optional beliebig viele Parameter an die Prozedur in ***'Para1, Para2%, Para3$, ...'*** übergeben werden. Dabei ist darauf zu achten, daß Anzahl, Typ und Listenplatz der Parameter mit Anzahl, Typ und Listenplatz der in der Prozedur benannten Aufnahmevariablen übereinstimmen (s. Beispiel zu **PROCEDURE**).

Die Angabe von @ bzw. GOSUB vor dem Prozedurnamen ist in den meisten Fällen sogar überflüssig. Es braucht nur der Prozedurname (gfls. mit der Parameterliste) angegeben zu werden (s. Beispiel zu **PROCEDURE**). Sind Verwechslungsmöglichkeiten mit Befehlsnamen möglich (z.B. `@Return` oder `@Run`), ist die Angabe von @ bzw. GOSUB allerdings zwingend notwendig.

GOTO { GOT } unbedingter Sprung zu einem Label

```
GOTO Label
```

Es wird zu einer beliebigen Programmstelle gesprungen, die vorher durch ein sogenanntes ***'Label'*** (*Programmarke*) zu definieren ist. GOTO-Sprünge in oder aus **FOR/NEXT**-Schleifen oder **PROCEDURE**s sind nicht möglich.

'Label' ist ein beliebiger Name, der zur Markierung eines Sprungziels für GOTO- oder **RESTORE**- Anweisungen dient. Er kann an jeder beliebigen Stelle im Programm placiert werden und ist durch einen nachgestellten Doppelpunkt zu kennzeichnen.

z.B.:

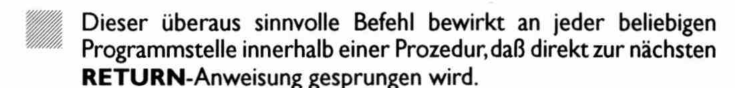
```
a=1
IF a THEN GOTO marke // Sprung zum Label
? '''a' war Null!''
marke:    // labelname 'marke:'
? '''a' ist ungleich Null!''
```

oder:

```
RESTORE marke
FOR i%=0 TO 4
  READ a
NEXT i%
marke:
DATA 1,2,3,4,5
```

In modernen BASIC-Dialekten, die - wie GFA-BASIC - auf der Grundidee der modularen und strukturorientierten Programmierung basieren, sollte der Befehl GOTO tunlichst vermieden werden. Seine Verwendung kennzeichnet den *'Spaghetti'*-Programmierer, der es anhand dieses unscheinbaren GOTO-Befehls spielend fertigbringt, selbst ein kleines 2 oder 5 Kbyte-Progrämmchen so unübersichtlich zu gestalten, das später bei evtl. notwendigen Änderungen niemand (noch nicht einmal er selbst) etwas damit mehr anzufangen weiß. Außerdem gilt es als erwiesen, daß gerade der GOTO-Befehl - und die mit ihm verbundene Denkweise - zu einer erheblichen Steigerung der logischen Fehlerquoten führt.

Der gerade Weg ist eben doch meist auch der kürzere - und das heißt: modulare Programmierung per **PROCEDURE** und **FUNCTION**.



EXPROC { EXP } unbedingter Sprung zum Prozedur-Ende

EXPROC

Dieser überaus sinnvolle Befehl bewirkt an jeder beliebigen Programmstelle innerhalb einer Prozedur, daß direkt zur nächsten **RETURN**-Anweisung gesprungen wird.

Statt einer IF-Abfrage in Form von:

```
PROCEDURE xyz
  IF bedingung wahr
    ...auszuführende Programmteile
  ENDIF
RETURN
```

kann die folgende Variante eingesetzt werden:

```
PROCEDURE xyz
  IF bedingung unwahr THEN EXPROC
  ...auszuführende Programmteile
RETURN
```

In **FUNCTION**s ist dieser Befehl nicht sinnvoll einsetzbar, da über den hier notwendigen ***'RETURN Rückgabe'***-Befehl hinweg



ohne Funktionsergebnis direkt zum aufrufenden Programm zurückgesprungen wird.

ON ... GOSUB bedingte Verzweigung zu Prozeduren

```
ON Wert GOSUB Proc1 [,Proc2,Proc3,...]
```

Verzweigt je nach übergebenem Wert zu einer der Listen-Prozeduren.

Die Verzweigung erfolgt nach folgendem Schema:

INT('Wert') = 1 -> verzweige zu Proc1
INT('Wert') = 2 -> verzweige zu Proc2
INT('Wert') = 3 -> verzweige zu Proc3
usw.

Ist ***'Wert'*** größer als die Anzahl der angegebenen Prozeduren oder Null, wird das Programm in der auf ON...GOSUB folgenden Zeile fortgesetzt. Bei sämtlichen ON/GOSUB-Varianten des GFA-BASICs (**ON ERROR GOSUB...** etc.) sind nur Prozeduren als Ziel erlaubt, die keine Parameterliste erwarten.

ON BREAK [CONT] [GOSUB] Break-Funktion behandeln

```
ON BREAK [GOSUB] Prozedur
ON BREAK
ON BREAK CONT
```

GFA-BASIC verfügt lobenswerter Weise über eine Funktion, die es erlaubt, das Programm an jeder beliebigen Programmstelle zu unterbrechen. Dazu werden gleichzeitig die Tasten <Strg> und <Untbr> (bzw. bei älteren Tastaturen <Ctrl> und <Break>) gedrückt. Diese Unterbrechungsfunktion kann durch den ON BREAK-Befehl behandelt werden.

Im ersten Fall (ON BREAK GOSUB...) verzweigt das Programm zu der angegebenen ***'Prozedur'***, sobald nach Ausführung dieses Befehls im Programmlauf die o.g. Break-Funktion angewandt wird. In der ***'Prozedur'*** kann dann entsprechend auf die Unterbrechung reagiert werden (GOSUB kann weggelassen werden).

Mit ON BREAK wird nach ON BREAK GOSUB... oder nach ON BREAK CONT die Break-Standardreaktion aktiviert. Das heißt, das Programm wird - wie direkt nach Programmstart - bei Betätigung der Tastenkombination <Strg><Untbr> sofort beendet.

ON BREAK CONT bewirkt dagegen, daß das Programm bis zum nächsten ON BREAK oder ON BREAK GOSUB bzw. bis zum Programmende nicht mehr durch die Break-Funktion unterbrochen werden kann. Diese Variante sollte während der Programmentwicklung mit Vorsicht genossen werden. Den meisten von Ihnen wird es sicher in unangenehmer Erinnerung sein, wenn sich ein Programm in einer Endlosschleife *'aufhängt'*, weil z.B. die Ausstiegsbedingung falsch formuliert war.

VAR — direkte Variablen-Übergabe

```
PROCEDURE Name([Var,...,]VAR Ref1[,Ref$,...])
FUNCTION Name([Var,...,]VAR Ref1[,Ref$,...])
```

Innerhalb der Kopfzeilen von **PROCEDURE**n und **FUNCTION**en können in der Liste der lokalen Aufnahme-Variablen (sog. *CallByValue*-Variablen) durch VAR auch Variablen direkt an die Prozedur/Funktion übergeben werden (sog. *CallByReference*-Variablen) übergeben werden.
Der in der VAR-Liste angegebene Variablenname steht dann innerhalb der **PROCEDURE** bzw. **FUNCTION** stellvertretend für die durch **GOSUB** bzw. **FN** übergebene Referenz-Variable. Es ist tatsächlich dieselbe Variable, die für die Dauer der Prozedur bzw. Funktion dann eben nur den in der Kopfzeile angegeben Namen trägt. Inhalte globaler Variablen mit demselben Namen bleiben dadurch unverändert, der prozedur- bzw. funktionsinterne Name hat also bis zum Rücksprung in das Hauptprogramm lokalen Charakter.

z.B.:

```
a|=10,b|=20
PRINT V:a|,V:b| // globale Variablen-Adressen
@Unterprog(a|)  // Variable a| direkt übergeben
PRINT a|''b|    // a| enthält nun den Wert 100
PROCEDURE Unterprog(VAR b|) // Die Variable a|
  //  wird jetzt innerhalb der Routine zu der
  //  lokalen Variable b|. Die globale Variable
  //  b| bleibt war erhalten, ist aber in der
  //  Prozedur nicht ansprechbar.
  b|=100        // b| wird lokal angesprochen
  PRINT V:b|    // Die Variablenadresse der
  //     lokalen Variable b| ist mit
  //     der Variablenadresse der
  //     globalen Variable a| identisch.
RETURN
```

Es ist dabei zu beachten, daß die beiden korrespondierenden Variablennamen im Prozedur-Aufruf und im Prozedur-Kopf dem gleichen Typ angehören (s. VARIABLEN-TYPEN).

Es ist auch möglich, Felder direkt an ein Unterprogramm zu übergeben:

```
x|=4                   // Feldgröße
DIM a&(x|)             // Feld dimensionieren
FOR i|=1 TO x|         // alle Elemente
  a&(i|)=i|            // mit Wert belegen
NEXT i|
@Unterprog(x|,a&())// Aufruf
FOR i|=1 TO x|         // alle Elemente
  PRINT a&(i|)         // neue Werte ausgeben
NEXT i|
PROCEDURE Unterprog(xx|,VAR aa&())// Feld-VAR !
  // Globalfeld a&() ist jetzt Feld aa&()
  FOR j|=1 TO xx|  // alle Elemente
    PRINT aa&(j|)  // alten Wert ausgeben
    aa&(j|)=j|*TRUE// mit neuem Wert belegen
  NEXT j|
RETURN
```

7.5. EXTERNE UNTERPROGRAMME UND INTERRUPTS

CALL { CAL } Maschinenprogramm aufrufen

```
CALL Adressvar% [(Parameterliste)]
CALL (Adresse) [(Parameterliste)]
```

'Adressvar%' ist eine 4Byte-Integervariable, welche die Startadresse der aufzurufenden Maschinenroutine (*assembliert* oder *C-compiliert*) enthält. Soll die Startadresse direkt oder als numerischer Ausdruck angegeben werden, so ist die ***'(Adresse)'*** in Klammern zu setzen. Beachten sie hierbei bitte die mögliche Adressverschiebung (s. Anmerkung unter **DIM**).

Bei dem Aufruf handelt es sich um einen sog. *'far call'*. Bei C-compilierten Programmen ist daher darauf zu achten, daß unter *'model large'* compiliert wird und daß generell die Maschinenroutine durch eine *'retf'*-Anweisung wieder in Richtung BASIC-Programm verlassen wird.

'Parameterliste' enthält optional, durch Kommata getrennt, die evtl. zu übergebenden Parameter. Diese werden wie bei **INTR()** den Registervariablen direkt zugewiesen.

z.B.

```
adresse%=$3ea:$0
CALL adresse%(_AL=$e6,_DI=1,_SI=$ff)
```

oder

```
_AX=$e6
_DI=1
_SI=$ff
CALL ($3ea:$0)
```

Stringparameter oder größere Parameterblöcke sind hier vorzugsweise mit ihren jeweiligen Anfangsadressen (z.B. Segment in **_AX** und Offset in **_BX**) zu übergeben.
Beim Rücksprung zum BASIC bleiben die aktuellen Register-Inhalte erhalten. Das heißt also, daß in den Registern **AX**, **BX** etc. auch Werte an das BASIC-Programm zurückgegeben und dort ausgelesen werden können.

C:() Maschinenprogramm nach C-Konvention aufrufen

```
~C:Adressvar% [(Parameterliste)]
Var=C:Adressvar% [(Parameterliste)]
```

Es ist in der 4Byte-Integervariablen ***'Adressvar%'*** die Adresse einer Maschinen-Routine und optional gfls. eine Liste numerischer Parameter in Klammern zu übergeben. Die Übergabe erfolgt nach üblichen C-Konventionen in umgekehrter Reihenfolge auf dem Stack. Der letzte Parameter wird also zuerst auf dem Stack abgelegt, wodurch der erste Parameter dann 'obenauf' liegt.

Sind keine Parameter zu übergeben, ist eine Leerklammer ***'()'*** zu verwenden. Sollen 32 Bit-Parameter übergeben werden, ist dem jeweiligen Parameter das Kürzel ***L:*** voranzustellen. Sonst gilt als voreingestelltes Parameterformat das Word (16 Bit).

z.B.

```
~C:(L:Para1%,Para2&,L:V:Para4$)
```

erzeugt folgende Stack-Situation:

```
push bp        ; bp retten
mov  bp,sp     ;   stackpointer nach bp laden
push di        ; di retten
push si        ; si retten
les  di,[bp+6] ;1. parameter 'L:Para1%'  (long)
mov  ax,[bp+10];2. parameter 'Para2&'    (word)
les  si,[bp+12];3. parameter 'L:V:Para3$'(long)
... etc.
... Maschinenprogramm
pop si         ; si rücksetzen
pop di         ; di rücksetzen
pop bp         ; bp rücksetzen
...
retf           ; nur 'retf' verwenden !!
```

Eine gfls. notwendige Restauration der Register bei Rücksprung obliegt dem Aufrufer bzw. sollte innerhalb der Maschinenroutine erfolgen.

Nach Rückkehr kann der Inhalt von **DX** (=Segment bzw. HI-Word) und **AX** (=Offset bzw. LOW-Word) als 4Byte-Wert entweder direkt ausgegeben (`PRINT C:Var%()`), einer Variablen übergeben (`A%=C:Var%()`) oder in Bedingungsabfragen und ähnlichen Konstruktionen eingebunden werden (`IF C:Var%()`). Bei dem Aufruf handelt es sich - wie bei **CALL** - um einen sog. *'far call'*. Beachten Sie dazu bitte die Anmerkung unter **CALL**.

INTR() MSDOS- oder BIOS-Interrupt aufrufen

```
~INTR(Nummer [,_AH=Unterfunktion,Reg=Wert,...]
```

Ein PC verfügt im allgemeinen über eine schier unüberschaubare Fülle an System-Funktionen und sonstigen Interrupt-Routinen. Mit INTR() ist es möglich, diese - oft sehr nützlichen - systeminternen Unterroutinen für eigene Programme nutzbar zu machen. Dazu wird in ***'Nummer'*** der übergeordnete Interrupt-Index und gfls. in Register **_AH** oder **_AX** die Nummer der gewünschten ***'Unterfunktion'***. Weitere benötigte Funktionsparameter können ebenfalls in der ***'Reg=Wert'***-Liste übergeben werden.

Interrupts benötigen vor ihrem Aufruf die Belegung verschiedener System-Register. Dies kann erfolgen, indem vor dem entsprechenden INTR()-Aufruf den betreffenden Registern mittels der dazu reservierten GFA-Variablen ein Wert zugewiesen wird:

```
_AH=8
_BH=1
~INTR(16)
```

oder:

```
_AH=8,_BH=1
~INTR(16)
```

oder:

```
_BH=1
~INTR(16,_AH=8)
```

Es ist jedoch auch möglich, eine Registerliste zu übergeben, in welcher die Registerbelegungen mit dem INTR()-Aufruf erledigt werden:

```
~INTR($33,_AX=4,_CX=100,_DX=100)
```

Bei vielen Interrupts werden in den verschiedenen Registern - je nach Interrupt variierend - auch Werte zurückgegeben. Diese können dann ebenfalls über die reservierten GFA-Registervariablen ausgelesen werden (z.B. `'Rückgabe'=_BL`).

Informationen über die verschiedenen Interrupts und die jeweils vorher zu belegenden Register, sowie die möglichen Rückgabe-Register finden Sie in der gängigen PC-Literatur. Ich verwende dazu das *'PC Intern 2.0'*-Buch von Michael Tischer (DATA BECKER),

das mir auch in anderen Fragen immer wieder wertvolle Dienste geleistet hat. Wie bereits im Vorwort angekündigt, wird ca. im Mai 1992 ein weiteres Buch zum ´GFA-BASIC für MSDOS´ vom COLID-Verlag erscheinen. In diesem Buch wird dann auch ein Auflistung der wichtigsten und nützlichsten DOS-, BIOS-, EGA/VGA- und EMS-Interrupts enthalten sein.

MONITOR {MON} Breakpoint-Interrupt $3 auslösen

```
MONITOR [(Parameter)]
```

Mit diesem Befehl kann innerhalb eines compilierten Programms der System-Interrupt **$3** (*Breakpoint-Interrupt*) ausgelöst werden. Dadurch ist es möglich, in einen Debugger zu verzweigen, um z.B. den Stand der Prozessor-Register zu überprüfen.

'Parameter' enthält optional einen Übergabewert, dessen LOW-Word bei Aufruf in **AX** und dessen HI-Word in **DX** eingetragen wird.

P:() Maschinenprogramm nach PASCAL-Konvention aufrufen

```
~P:Adressvar% ([Parameterliste])
Var=P:Adressvar% ([Parameterliste])
```

Es gelten exakt die gleichen Erläuterungen wie zu C:, nur daß hier die Parameter ihrer Reihenfolge nach von links nach rechts auf den Stack gelegt werden, wodurch dann - im Gegensatz zu C: - der letzte Parameter 'obenauf' liegt.

z.B.

```
~P:(L:Para1%,Para2&,L:V:Para4$)
```

erzeugt folgende Stack-Situation:

```
...  ; bis hierher wie im C:-Beispiel
...
les  si,[bp+6] ; 3. parameter 'L:V:Para3$'(long)
mov  ax,[bp+10]; 2. parameter 'Para2&'    (word)
les  di,[bp+12]; 1. parameter 'L:Para1%'  (long)
...
...  ; ab hier wieder wie im C:-Beispiel
```

Alles weitere lesen Sie bitte bei der Beschreibung des **'C:'**-Befehls nach.

7.6. REGISTER - VARIABLEN

_AX, _AH, _AL	('A'ccu) AX-Register-Word
_BX, _BH, _BL	('B'ase) BX-Register-Word
_CX, _CH, _CL	('C'ount) CX-Register-Word
_DX, _DH, _DL	('D'ata) DX-Register-Word

```
Var=_AX                  -> AX-Word komplett lesen
_AX=Wordwert             -> AX-Word komplett schreiben

 Var=_AH                 -> HiByte des AX-Words lesen
_AH=Bytewert             -> HiByte des AX-Words schreiben

Var=_AL                  -> LoByte des AX-Words lesen
_AL=Bytewert             -> LoByte des AX-Words schreiben
```

Bei diesen Register handelt es sich um die allgemeinen System-Register der Intel-80xxx-Prozessoren. Es kann hierüber jeweils das komplette Word des entsprechenden Registers oder separat entweder sein Low-Byte oder sein High-Byte angesprochen werden (s. auch unter **CALL** und **INTR**).

In den Syntax-Zeilen sind die AX-/AH-/AL-Modifikationen als Beispiele angegeben. Bei den anderen Register-Variablen (_BX, _BL, _BH, _CX etc.) ist analog zu verfahren.

_DI	Destination-Index-Word
_SI	Source-Index-Word
_BP	Base-Pointer-Word
_FL	Flag-Register-Word
_SP	Stack-Pointer-Word

```
Var=_DI                  -> DI-Word komplett lesen
_DI=Wordwert             -> DI-Word komplett schreiben
```

Außer den reinen Daten-Registern (**_AX**, **_BX** etc.) verfügen die Intel-Prozessoren noch über verschiedene Register zur speziellen Verwendung. Die Verwendung und Inhalt dieser Register ist vom jeweiligen Fall abhängig (s. auch unter **CALL** und **INTR**).

In der Syntax-Beschreibunge ist die DI-Modifikationen als Beispiel angegeben. Bei den anderen Sonder-Registern (_SI, _BP, _FL, _SP) ist analog zu verfahren.

_EAX, _EBX, _ECX, _EDX 386/486er Register-Longs
_EDI, _ESI, _EBP, _EFL, _ESP 386/486er Register-Longs

```
Var=_EAX          -> EAX-HiWord lesen
_EAX=Wert         -> EAX-HiWord schreiben
```

Dieses sind die erweiterten (engl.: *extended*) System-Register der neueren *386er* und *486er* Intel-Prozessoren. Es können hierüber jeweils die High-Words des entsprechenden Registers angesprochen werden.

In der Syntax-Zeile ist die EAX-Modifikation als Beispiel angegeben. Bei den anderen Registern-Variablen (_EBX, _ECX etc.) ist analog zu verfahren.

7.7. AUSFÜHRBARE PROGRAMME

EXEC { EXE } / EXEC() COM/EXE-Programm laden/starten

```
EXEC ''Programmname'',''Kommandozeile''
Var=EXEC(''Programmname'',''[Kommandozeile]'')
```

EXEC lädt und startet ein komplett lauffähiges MSDOS-Programm, wobei das aufrufende Programm (GFA-Compilat, GFA-Interpreter) resident im Speicher bleibt und das aufgerufene Programm dorthin wieder zurückkehrt.

Die erste Syntaxform (als Befehl) ist nur dann sinnvoll, wenn kein Rückgabewert vom aufgerufenen Programm erwartet wird. Anderenfalls kann mit der zweiten Variante (EXEC() als Funktion) ein Rückgabewert (z.B. Fehlercode, Zeiger etc.) vom aufgerufenen Programm empfangen werden.

Vor der EXEC-Ausführung sollte dafür gesorgt werden, daß genügend Speicherplatz für die Anwendung zur Verfügung steht. Gfls. sind große Arrays und Strings vorher in den EMS-Speicher zu verlegen.

'Programmname' enthält den vollständigen Namen (gfls. incl. Pfad) des zu ladenden und zu startenden Programms.

'Kommandozeile' enthält eine evtl. zu übergebende Kommandozeile, die an der entsprechenden Position im *'Programmsegment Präfix'* (s. **_PSP**) des aufgerufenen Programms abgelegt wird. Soll dieser Parameter nicht übergeben werden, kann dafür ein Leerstring ("") verwendet werden. Speicherresidente Programme sollten hiermit nicht aufgerufen werden, da eine nachträgliche Freigabe des verwendeten Programmspeichers dann nicht mehr möglich ist.

SHELL { SH } 'COMMAND.COM' starten/DOS-Befehl ausführen

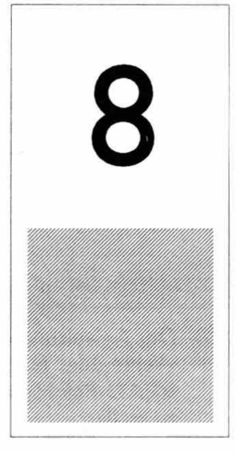
```
SHELL ''[DOS-Befehl]''
```

Dieser Befehl ermöglicht den Aufruf des MSDOS-Kommando-Interpreters **'COMMAND.COM'**. Wird dabei die Option ***'DOS-Befehl'*** nicht verwendet, so erscheint das *DOS-Prompt* und man kann nun verschiedene Arbeiten auf DOS-Ebene ausführen. GFA-BASIC bleibt resident und man kann den DOS-CLI durch den **EXIT**-Befehl wieder in Richtung BASIC verlassen.

Wird dagegen innerhalb der Anführungsstriche ein MSDOS-Befehl angegeben, so wird der Befehl direkt auf DOS-Ebene ausgeführt und anschließend automatisch zum BASIC zurückgekehrt.

Notizen:

Daten-Organisation



8. DATEN - ORGANISATION

8.1. BEREICHS - DEKLARATIONEN

// Kommentarbeginn am Ende einer Befehlszeile

```
...Programmtext...// [Kommentar]
```

Es kann zum Abschluß einer beliebigen Programmzeile (außer **DATA**) ein Kommentar angehängt werden. Die auf // folgenden Zeichen bleiben bei der Programmausführung unberücksichtigt.

// kann auch am Zeilenbeginn (s. **REM**) eingesetzt werden.

/*...*/ Kommentarkennung innerhalb einer Befehlszeile

```
Programm.../*[Kommentar]*/...Programm
```

Es kann innerhalb einer beliebigen Programmzeile (außer **DATA**) ein Kommentar eingefügt werden. Die zwischen der Anfangskennung ***/** und der Endekennung **/*** stehenden Zeichen bleiben bei der Programmausführung unberücksichtigt.

REM { R oder ' } Kommentar einfügen

```
REM [Kommentar]
' [Kommentar]
```

Es kann eine beliebige Programmzeile als Kommentarzeile deklariert werden. Die darin enthaltenen Zeichen bleiben bei der Programmausführung unberücksichtigt.

DATA { D } Daten-Speicher

```
DATA [Num.Daten [,['‘] Textdaten ['‘],...]]
```

Der Anweisung wird gfls. eine Liste, durch Kommata getrennter Werte oder Texte übergeben. Sie dient dazu, einem auftretenden **READ**-Befehl die entsprechende Anzahl von Daten zur Verfügung zu stellen.

z.B.:

```
DATA 12,$FF00,ABC,''Text, Text''
READ a|,b&,tx1$,tx2$
```

Wenn in Text-Datas keine Kommata berücksichtigt werden müssen, brauchen diese Texte nicht in Anführungszeichen gesetzt zu werden. Der Interpreter liest in diesem Fall alle Zeichen (auch Leerzeichen), die zwischen den einschließenden Kommas aufgeführt sind. Numerische **READ**-Anweisungen sind darauf angewiesen, auch numerische Datas vorzufinden. Die Datas können dann allerdings auch in der binären, hexadezimalen oder oktalen Schreibweise angegeben sein.

READ { REA } — DATA-Werte auslesen

```
READ Var [,Var2,Var3%,Var4$,...]
```

Den angegebenen Variablen ***'Var'*** werden die jeweils gelesenen **DATA**-Einträge zugeordnet. Wird kein **RESTORE'Label'** verwendet, werden der Reihe nach vom Programmanfang aus soviele Datas eingelesen (falls vorhanden), wie READ-Anweisungen ausgeführt werden.

RESTORE { RES } — DATA-Zeiger setzen

```
RESTORE [Label]
```

RESTORE ohne Angabe eines Labels bewirkt, daß der **READ**-Zeiger auf die erste **DATA**-Zeile im Programm gerichtet wird. Die folgenden **READ**-Anweisungen beziehen ihre Daten nacheinander ab dieser Zeile. Wird dem Befehl ein ***'Label'*** übergeben, zeigt der **READ**- Zeiger auf den Anfang der **DATA**-Zeile, die auf das angegebene ***'Label'*** folgt. Ein Label ist eine Programm-Marke, durch welche eine bestimmte Stelle innerhalb des Programms identifiziert werden kann. Diese Marke besteht aus einer (fast) beliebigen Anordnung von Textzeichen und Ziffern, die durch einen Doppelpunkt abgeschlossen werden müssen.

z.B.:

```
RESTORE d_label /* setzt DATA-Zeiger auf 'd_label'
READ txt$,num%  /* liest String und Wert       '
PRINT txt$'num% /* gibt 'xyz 99' aus
DATA ''abc'',1  /* 1. DATA-Zeile (wird ignoriert)
d_label:        /* Programm-Markierung ('Label')
DATA ''xyz'',99 /* 2. DATA-Zeile
```

_DATA interne Variable für DATA-Zeiger

```
_DATA=Adresse Var=_DATA
```



Während der **RESTORE**-Befehl ein Setzen des **DATA**-Zeigers auf den Anfang einer bestimmten **DATA**-Zeile ermöglicht, kann man durch die reservierte Variable _DATA diesen Zeiger auch direkt auf eine bestimmte Position innerhalb einer **DATA**-Zeile lenken, bzw. die aktuelle Position des **DATA**-Zeigers ermitteln.

Da sich diese Position relativ zur Programmlänge und Lage der **DATA**-Zeilen dynamisch verhält (und sich also ändern kann), müssen die später evtl. durch _DATA= anzuspringenden **DATA**-Positionen vorher (gfls. bei Programmstart) durch entsprechende **READ**-Anweisungen mit anschließender _DATA-Abfrage ermittelt und zwischengespeichert werden. Danach können diese **DATA**-Positionen bis zum Programmende als absolut, also unveränderlich angesehen werden.

_DATA enthält den **DATA**-Zeiger in Form eines 4Byte-Longs, der auf eine bestimmte Speicherposition innerhalb des Programm-(*Token-*) Codes weist. An dieser Speicherposition befindet sich dann - generell im ASCII-Format - das dazugehörige **DATA**-Element.

8.2. FELDER UND ARRAYS

ARRAYFILL { ARR } Feld mit Wert belegen

```
ARRAYFILL Feld(),Wert
```



'Feld' bezeichnet ein dimensioniertes, numerisches oder bool'sches Feld. Alle Elemente dieses Feldes werden mit dem angegebenen ***'Wert'*** belegt.

Paßt das ***'Wert'***-Format nicht mit dem Feldtyp überein (z.B. ***'Feld()'*** ist Byte-Format und ***'Wert'*** ist Real), so wird nur der Anteil von ***'Wert'*** berücksichtigt, der für den Typ von ***'Feld()'*** verwendbar ist (z.B. `INT(Wert) AND $FF`). Bei Bool-Feldern bewirkt jeder ***'Wert'***, der ungleich Null ist, ein Füllen mit **TRUE** (-1).

DELETE { DEL } Einzelelement aus Feld löschen

```
DELETE Feld(Index)
DELETE Feld$(Index)
```

Löscht das einzelne Element ***'Index'*** aus dem eindimensionalen ***'Feld()'***, bzw. ***'Feld$()'***. Alle darüberliegenden Elemente werden im Array um eine Stelle nach unten versetzt. Das letzte Element des Arrays enthält anschließend den Wert Null, bzw. bei String-Arrays einen Leerstring ("").

DIM { DI } Feld(er) dimensionieren

```
DIM Feld1(D1[,D2,..])[,Feld2(D1[,D2,..])..]
```

Legt die Dimension(en) von ***'Feld1()'*** (bzw. gfls. ***'Feld2()'***, ***'Feld3()'*** etc.) fest und reserviert hierfür Speicherplatz. Dabei sind pro Feld bis zu 6 Dimensionen möglich.

'Feld()' steht für beliebige numerische oder String-Felder. ***'Dx'*** besagt, wieviele Elemente pro Dimension eingerichtet werden sollen. Bei mehrdimensionalen Feldern darf die Anzahl der Elemente in der ersten Dimension nicht größer als 65535, und die gesamte Anzahl der Elemente der übrigen Dimensionen (das Produkt der Dimensionen 2 bis 6) ebenfalls nicht größer als 65535 sein. Bei eindimensionalen Feldern richtet sich die Feldgröße dagegen nur nach der Größe des verfügbaren Arbeitsspeichers.

ACHTUNG:

Bei sehr großen Dimensionierungen kann sich die Adressenlage der übrigen Variablen - insbesondere bei Stringvariablen - verschieben. Im Falle, daß Maschinenprogramme in Stringvariablen abgelegt wurden, führt dies beim nächsten Aufruf der Routine gfls. zum Absturz. Zur Speicherung der Routine eignet sich daher am besten ein beliebiges numerisches Feld:

```
DIM A%(Codelänge/4+1)
Start%=VARPTR(A%(0))
BLOAD ''MASCHINE.COD'',Start%
CALL Start%
```

DIM?() Menge der Feldelemente ermitteln

```
Var=DIM?(Feld())
```

'Feld()' ist ein beliebiges numerisches oder String-Feld. DIM?() liefert die Anzahl aller Elemente dieses Feldes. Bei nicht dimensionierten Feldern wird der Wert 0 geliefert.

ERASE { ER } Feld(er) löschen

```
ERASE Feld1() [,Feld2() [,...]]
```

'Feld()' bezeichnet ein beliebiges Feld, das gelöscht werden soll. Die Dimensionierung für dieses Feld wird aufgehoben und der dafür reservierte Speicherplatz wieder freigegeben. Es ist möglich, auch eine Liste von Feldern (auch verschiedner Typen) anzugeben, die dann mit nur einem Befehl gelöscht werden.

INSERT { INS } Einzelelement in Feld einfügen

```
INSERT Feld(Index)=Wert
INSERT Feld$(Index)=''Text''
```

Fügt das einzelne Element ***'Index'*** in das eindimensionale ***'Feld()'***, bzw. ***'Feld$()'*** mit dem zugewiesenen ***'Wert'*** bzw. ***'Text'*** ein. Alle darüberliegenden Elemente werden um eine Stelle nach oben versetzt. Der Inhalt des letzten Elementes von ***'Feld()'*** bzw. ***´Feld$()´*** wird dabei überschrieben.

OPTION BASE { OPT B } Feld-Basiselement bestimmen

```
OPTION BASE 0
OPTION BASE 1
```

Bestimmt das Basis-Element aller dimensionierten Felder (0 oder 1). Die Basis (das Element mit dem kleinsten Index) kann im Programm mehrmals geändert werden, da sich die schon definierten Elemente dem neuen Index anpassen. Z.B. vorher OPTION BASE 0, dann OPTION BASE 1: aus A$(0) wird A$(1), A$(1) wird A$(2) etc. War das Feld() vorher mit z.B. `DIM Feld(10)` eingerichtet, existiert dann auch das Element Feld(11).

QSORT { Q } Feld (-Bereich) Quick-Sortierung

```
QSORT Feld([+/-]) [,Anz [,Feld2%()]]
QSORT Feld$([+/-])[WITH Sort()][,Anz[,Feld2%()]]
```

Es können Felder nach ihrer numerischen Größe oder alphabetischer Reihenfolge nach dem *'Quicksort'*-Verfahren sortiert werden. ***'Feld()'*** ist dabei ein numerisches Feld beliebigen Typs. ***'Feld$()'*** ist ein Stringfeld. ***'+/-'*** (innerhalb der Leerklammer, z.B. QSORT Feld(+)) ist entweder ein Plus- oder Minuszeichen, daß die Sortierrichtung angibt. Sollen die Werte, bzw. Strings mit dem höchsten Wert, bzw. Buchstaben im niedrigsten Element (0 bei **OPTION BASE 0)** beginnend absteigend sortiert werden, ist dies das Minuszeichen **'-'**. Ein Pluszeichen **'+'** oder keine Angabe bewirkt die aufsteigende Sortierung (niedrigster Wert bzw. Buchstabe im niedrigsten Element).

'Anz' kann optional verwendet werden, um zu bestimmen, wieviele Elemente maximal sortiert werden soll (z.B. 6 = von 0-5 bei **OPTION BASE 0**, bzw. von 1-6 bei **OPTION BASE 1**). Es kann optional ein 4Byte-Integerfeld (***'Feld2%()'***) angegeben werden, dessen Elemente unabhängig von ihrem Inhalt paralell mit dem eigentlichen Sortierfeld mitsortiert werden. Steht nach der Sortierung z.B. der Inhalt des vorherigen ***'Feld()'***-Elementes mit dem Index 6 nun im Element-Index 3, so wird der Inhalt des vorherigen ***'Feld2%()'***-Elementes 6 nun unabhängig von seinem Wert ebenfalls im Element-Index 3 einsortiert sein.

Bei Stringfeldern kann durch ***WITH*** zusätzlich ein beliebiges Integerfeld (***'Sort()'*** = 1Byte, 2Byte oder 4Byte-Integer) mit mindestens 256 Elementen bestimmt werden, dessen Elemente-Inhalt die Reihenfolge der Sortierung vorgibt. Sind z.B. alle 256 Elemente mit den ASCII-Werten in normaler Reihenfolge belegt (0-255), ist die Verwendung von ***'Sort()'*** überflüssig, da die ASCII-Tabelle defaultmäßig als Sortierfolge angenommen wird. Werden dagegen z.B. die Zeichen *'a'* und *'A'* vertauscht, steht *'A'* in der Sortierfolge über *'a'* (normalerweise umgekehrt), während alle anderen Zeichen normal sortiert werden. So kann eine völlig willkürliche Sortierfolge vorgegeben werden.

SSORT { SS } Feld (-Bereich) Shell-Sortierung

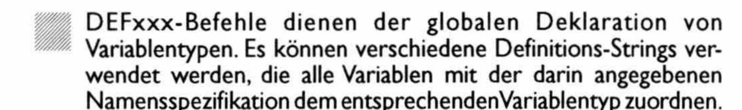
```
SSORT Feld([+/-]) [,Anz [,Feld2%()]]
SSORT Feld$([+/-])[WITH Sort()][,Anz[,Feld2%()]]
```

Erläuterungen zu **QSORT** gelten hier analog (s. dort).

8.3. VARIABLEN - DEKLARATION

DEFBIT { DEFBI } — Boolvariable(n) deklarieren

```
DEFBIT Define$
```



DEFxxx-Befehle dienen der globalen Deklaration von Variablentypen. Es können verschiedene Definitions-Strings verwendet werden, die alle Variablen mit der darin angegebenen Namensspezifikation dem entsprechendenVariablentyp zuordnen.

Diese Deklaration wird üblicherweise zu Programmbeginn ausgeführt, da sonst eine Unterscheidung zwischen deklarierten und frei definierten Variablen erschwert wird. Nach dieser Deklaration sieht das Programm alle Variablen, denen kein Postfix (z.B. **'%'** für 4Byte-Integers, **'!'** für Bool-Variablen oder **'#'** für Fließkomma-Variablen) angehängt wurde, als Variablen des angegebenen Typs an.

Es ist trotzdem jederzeit möglich, einzelne Variablen separat zu definieren, auch wenn sie dieselbe Namensspezifikation aufweisen wie eine globale Deklaration. Dazu ist in den entsprechenden Fällen dem separaten Variablennamen das entsprechende Postfix hinzuzufügen, womit die so unmißverständlich gekennzeichnete Variable von der Deklaration ausgeschlossen wird. Separat gekennzeichnete Variablen haben generell Vorrang vor den globalen Deklarationen. Um Mißverständnisse zu vermeiden, sollte nach Verwendung von DEFxxx am Programmende `DEFFLT ''a-z''` verwendet werden.

'Define$' ist ein String (Konstante, Variable oder Ausdruck), durch welchen die Namensspezifikation festgelegt wird.

Beispiele:

DEFBIT "a" = Variablen, deren Name als ersten Buchstaben ein 'a' tragen, sind hiermit - sofern nicht separat anders bestimmt (s.o.) - als Boolvariablen deklariert.

DEFWRD "word" = Variablen, deren Name mit den Buchstaben ***'word'*** beginnen, sind hiermit als 2Byte-Integervariablen deklariert.

DEFSTR "d-f" = Variablen, deren Name als ersten Buchstaben ein ***'d'***, ***'e'*** oder ***'f'*** tragen, werden als Stringvariablen interpretiert.

DEFBYT "b,c,g-l" = Variablen, deren Name als erstes Zeichen ein ***'b'***, ***'c'***, ***'g'***, ***'h'***, ***'i'***, ***'j'***, ***'k'*** oder ***'l'*** trägt, werden als *1Byte-Integervariablen* angesehen.

DEFINT "i1,i2" = Variablen, deren Name mit den Buchstaben ***'i1'***, oder ***'i2'*** beginnen, werden als *4Byte-Integervariablen* angesehen.

DEFFLT "a-c,x-z" = Variablen, deren Name als ersten Buchstaben ein ***'a'***, ***'b'***, ***'c'***, ***'x'***, ***'y'*** oder ***'z'*** trägt, werden hiermit als *8Byte-Fließkommavariablen* (Standard) deklariert.

DEFBYT { DEFB } 1Byte-Integervariablen deklarieren

```
DEFBYT Define$
```

Siehe Erläuterungen zu **DEFBIT**.

DEFDBL { DEFD } 8Byte-Fließkommavariablen deklarieren

```
DEFDBL Define$
```

Identisch mit **DEFFLT**. Siehe Erläuterungen zu **DEFBIT**.

DEFFLT { DEFFL } 8Byte-Fließkommavariablen deklarieren

```
DEFFLT Define$
```

Identisch mit **DEFDBL**. Siehe Erläuterungen zu **DEFBIT**.

DEFINT { DEFI } 4Byte-Integervariablen deklarieren

```
DEFINT Define$
```

Siehe Erläuterungen zu **DEFBIT**.

DEFSNG { DEFS } 4Byte-Fließkommavariablen deklarieren

```
DEFSNG Define$
```

Siehe Erläuterungen zu **DEFBIT**.

DEFSTR { DEFST } Zeichenkettenvariable(n) deklarieren

```
DEFSTR Define$
```

Siehe Erläuterungen zu **DEFBIT**.

DEFWRD { DEFW } 2Byte-Integervariablen deklarieren

```
DEFWRD Define$
```

Siehe Erläuterungen zu **DEFBIT**.

8.4. DATEN - UMWANDLUNG

BIN$() Numerisch => Binär

```
Var$=BIN$(Expr [,Stellen])
```

Wandelt ***'Expr'*** zu einem Textstring im Binärformat um. ***'Expr'*** steht für eine beliebige numerische Variable oder Ausdruck. Will man Integerwerte im Binär-Format angeben, so kann der Vorsatz ***'&X'*** (z.B.: `A%=&X10011101`) verwendet werden.

Durch den optionalen Parameter ***'Stellen'*** kann eine Stellenanzahl (1 - 32) vorgegeben werden, auf die der gewandelte Wert begrenzt oder durch vorangestellte Nullen erweitert wird.

DEC$() Numerisch => Dezimal

```
Var$=DEC$(Expr[,Stellen])
```

Wandelt ***'Expr'*** zu einem Textstring im Integer-Dezimalformat um. ***'Expr'*** steht für eine beliebige numerische Variable oder Ausdruck.

Durch den optionalen Parameter ***'Stellen'*** kann eine Stellenanzahl (1 - 32) vorgegeben werden, auf die der gewandelte Wert begrenzt oder durch vorangestellte Nullen erweitert wird.

HEX$() — Numerisch => Hexadezimal

```
Var$=HEX$(Expr [,Stellen])
```

Wandelt ***'Expr'*** zu einem Textstring im Hexadezimalformat um. ***'Expr'*** steht für eine beliebige numerische Variable oder Ausdruck. Will man Integerwerte im Hexadezimal-Format angeben, so kann der Vorsatz ***'&H'*** (z.B.: `A%=&HE1A7`) verwendet werden. Durch den optionalen Parameter ***'Stellen'*** kann eine Stellenanzahl (1 - 8) vorgegeben werden, auf die der gewandelte Wert begrenzt oder durch vorangestellte Nullen erweitert wird.

OCT$() — Numerisch => Oktal

```
Var$=OCT$(Expr [,Stellen])
```

Wandelt ***'Expr'*** zu einem Textstring im Oktalformat um. ***'Expr'*** steht für eine beliebige numerische Variable oder Ausdruck. Will man Integerwerte im Oktal-Format angeben, so kann der Vorsatz ***'&O'*** (z.B.: `A%=&O16501`) verwendet werden. Durch den optionalen Parameter ***'Stellen'*** kann eine Stellenanzahl (1 - 11) vorgegeben werden, auf die der gewandelte Wert begrenzt oder durch vorangestellte Nullen erweitert wird.

ASC() — Textzeichen => ASCII-Wert

```
Var=ASC(''Zeichen'')
```

Ermittelt den ASCII-Wert von ***'Zeichen'***. Bei Strings wird nur der ASCII-Wert des ersten Zeichens zurückgegeben. Ist der angegebene String leer (""), wird der Wert Null geliefert. ASC() bildet die Umkehrfunktion zu **CHR$()**.

CHR$() ASCII => Textzeichen

```
Var$=CHR$(Wert)
```

Liefert das, dem angegebenen *'Wert'* entsprechende ASCII-Zeichen. Ist *'Wert'* größer als 255, so wird das Zeichen ermittelt, das *'Wert' MOD 256* (bzw. *'Wert' AND $FF*) entspricht (s. Beispiel zu **XLATE$** und **SELECT..CASE**). CHR$() bildet die Umkehrfunktion zu **ASC()**.

Beachten Sie auch das Beispiel zu **STR$()**.

CVD() CVI() CVL() CVS() String => Format-Zahl

```
CVI(''2 Zeichen'')        ->    16 Bit-Integerzahl
CVL(''4 Zeichen'')        ->    32 Bit-Integerzahl
CVS(''4 Zeichen'')        ->    IEEE-Single-Realzahl
CVD(''8 Zeichen'')        ->    IEEE-Double-Realzahl
```

Es werden die, der jeweiligen Funktion entsprechenden, ersten *'x'* Zeichen des beliebigen Textstrings *"x Zeichen"* in eine Zahl des jeweiligen Formats umgewandelt. Ist *"x Zeichen"* kürzer als die für die jeweilige Funktion erforderliche Zeichenanzahl, so wird Null geliefert.

Diese Funktionen bilden die Umkehrfunktionen zu **MKD$(),MKI$(),MKL$(),MKS$()**. Beachten Sie auch die Ausführungen unter 'VARIABLEN - TYPEN' sowie unter **INT{}**, **LONG{}**, **SINGLE{}** und **DOUBLE{}**.

MKD$() MKI$() MKL$() MKS$() Format-Zahl => String

```
MKD$(IEEE-Double-Wert)      ->    8-Zeichenstring
MKI$(16Bit-Wert)            ->    2-Zeichenstring
MKL$(32Bit-Wert)            ->    4-Zeichenstring
MKS$(IEEE-Single-Wert)      ->    4-Zeichenstring
```

Es wird der in Klammern angegebene Wert in einen, der Wertgröße und dem gewünschten Format entsprechenden Stringausdruck umgewandelt.

Diese Funktionen bilden die Umkehrfunktionen zu **CVD(),CVI(),CVL(),CVS()**. Beachten Sie auch die Ausführun-

gen unter 'VARIABLEN - TYPEN' sowie unter **INT{}**, **LONG{}**, **SINGLE{}** und **DOUBLE{}**.

STR$() — Numerisch -> String

```
Var$=STR$(Wert [,Stellen,Realteil])
```



Es wird ein Textstring mit der Länge gebildet, die der Anzahl der Ziffern des übergebenen Wertes im Dezimalformat entsprechen würde.

'Wert' kann in jedem beliebigen Zahlensystem angegeben werden. Als *Hexadezimal-*, *Binär-* oder *Oktalzahl* angegebene Werte werden vorher in das Dezimalformat umgewandelt.

Durch die optionalen Parameter ***'Stellen,Realteil'*** kann eine gesamte ***'Stellen'***-Anzahl bestimmt werden (Vor- und Nachkommastellen incl. Dezimalpunkt), auf die der gewandelte Wert in der Länge begrenzt wird und die Anzahl an Stellen davon, die für den ***'Realteil'*** (Nachkommastellen) verwendet werden sollen.

z.B.:

```
PRINT STR$(572.6169,6,3)    ergibt:   72.617
```

STR$() bildet die Umkehrfunktion zu **VAL()**.

Beispiel:

```
SCREEN 18     /* oder 16 oder 14   // VGA oder EGA
DEFFILL 8                          // vollflächig
PRINT ''Weiter = li.Maustaste''
PRINT ''Ende   = re.Maustaste''
REPEAT
  wert=RANDOM(10^RAND(12))/(10^(RAND(6)+1))
  '                                // Zufalls-Real
  wert=wert-(wert*2*(RAND(2)))     // Zufallsvorzeichen
  dlen%=(LEN(STR$(wert))*40/1.5)   // Länge der
  '                                // Ziffernausgabe
  COLOR 15
  PBOX 25,50,615,110               // Hintergrund weiß
  COLOR 0
  BOX _X/2-dlen%/2-5,55,_X/2+dlen%/2+5,105 // Rahmen
  digit(wert,_X/2-dlen%/2,60,40,RAND(15),RAND(15))
  '                                // Aufruf
    REPEAT                         // warten...
    UNTIL MOUSEK                   // ...auf Maustaste
  UNTIL MOUSEK=2                   // re. Taste=Exit
```

```
PROCEDURE digit(num,gxs,gys,ho,icol,ocol)
  '
  ' Parameter:
  '----------
  ' num        = zu zeichnender Wert
  ' gxs, gys   = X-Start, Y-Start
  ' ho         = Zeichenhöhe in Pixelel
  ' icol, ocol = Rahmen- und Füllfarbe
  '
  LOCAL dig,gsc,j%,pat$
  pat$=CHR$(215)+CHR$(68)+CHR$(190)//- setzen der
  pat$=pat$+CHR$(238)+CHR$(109)  //   Ziffern-Module
  pat$=pat$+CHR$(235)+CHR$(251)  //      für 0-9,
  pat$=pat$+CHR$(70)+CHR$(255)   //       sowie
  pat$=pat$+CHR$(239)+CHR$(64)   //    für Dez.punkt
  pat$+pat$+CHR$(40)             //- und Minuszeich.
  FOR j%=0 TO LEN(STR$(num))-1 //Wertstring durchgehen
    IF MID$(STR$(num),j%+1,1)=''.''// Dez.-punkt ?
      dig=10                     // ID für Punkt
    ELSE IF MID$(STR$(num),j%+1,1)=''-''// Minus ?
      dig=11                     // ID für Minus
    ELSE                         // sonst
      dig=VAL(MID$(STR$(num),j%+1,1))// ID für Ziffer
    ENDIF
    bit=ASC(MID$(pat$,dig+1,1))  // Modul-DefBits
    gsc=240/ho                // Scalierungsfaktor
    ' ab hier werden die jeweils notwendigen
    ' Ziffernmodule gezeichnet
    IF bit AND 1 THEN digmodul(gxs,gys+110/...
                      ... gsc,0,gsc,icol,ocol)
    IF bit AND 2 THEN digmodul(gxs+10/...
                 ...gsc,gys,90,gsc,icol,ocol)
    IF bit AND 4 THEN digmodul(gxs+120/gsc,...
            ...gys+10/gsc,180,gsc,icol,ocol)
    IF bit AND 8 THEN digmodul(gxs+110/...
        ...gsc,gys+120/gsc,270,gsc,icol,ocol)
    IF bit AND 16 THEN digmodul(gxs,gys+230/...
                      ...gsc,0,gsc,icol,ocol)
    IF bit AND 32 THEN digmodul(gxs+10/...
        ...gsc,gys+120/gsc,90,gsc,icol,ocol)
    IF bit AND 64 THEN digmodul(gxs+120/...
        ...gsc,gys+130/gsc,180,gsc,icol,ocol)
    IF bit AND 128 THEN digmodul(gxs+110/...
        ...gsc,gys+240/gsc,270,gsc,icol,ocol)
    gxs=gxs+160/gsc           // Ziffern-X-Offset
  NEXT j%
RETURN
PROCEDURE digmodul(xs,ys,dg,sc,ci,co)
  '
  ' Parameter:
  '----------
  ' xs, ys = X-Start, Y-Start
  ' dg     = Rotationswinkel
  ' sc     = Scalierungsfaktor
  ' ci, co = Rahmen- und Füllfarbe
  '
  COLOR ci                       // Rahmenfarbe
  DRAW ''ma'',xs,ys,''pd tt'',dg  //-----------
  DRAW ''fd'',100/sc,''rt 135"    // Turtle-DRAW
```

```
    DRAW ''fd'',40/sc,''rt 45"    //   für das Digit-
    DRAW ''fd'',43.5/sc,''rt 45" //   Grundmodul
    DRAW ''fd'',40/sc            //------------
    IF sc<11             // wenn Zeichen groß genug
      COLOR co           // Füllfarbe
      FILL xs+SINQ(160-dg)*50/sc,ys+COSQ(160-dg)*...
                                       ...50/sc,ci
    ENDIF                // Modul ausfüllen
  RETURN
```

Einige Zeilen im obigen Listing wurden aus drucktechnischen Gründen geteilt (Zeilenanfang... ...Zeilenrest)

Zu diesem Beispielprogramm ist technisch weiter nicht viel zu sagen. Erklärenswert wäre allerdings die logische Anordnung der Modul-*DefBits*:

Dazu betrachten Sie bitte eine durch *'digit'* gezeichnete digitale Acht. Sie werden feststellen, daß diese Acht aus acht Grundmodulen in verschiedenen Drehwinkel und Abständen aufgebaut ist. Das Grundmodul sieht ungefähr wie folgt aus:



Nun habe ich einfach für jede mögliche Ziffer einen 8Bit-Vektor eingerichtet, der anhand der Bitstellung bestimmt, welche Module jeweils zu zeichnen sind. Dabei bin ich davon ausgegangen, daß die Bits 0-3 jeweils die Module der oberen Ziffernhälfte - im Uhrzeigersinn vom Grundmodul links ausgehend - und die Bits 4-7 in der gleichen Reihenfolge die Module der unteren Ziffernhälfte repräsentieren. Ist ein Bit im Vektor gesetzt, wird das entsprechende der acht möglichen Ziffernmodule gezeichnet.

Diese Bit-Informationen stecken nun in dem 12 Zeichen langen Modul-Defstring *'pat$'*. Dabei entsprechen die Zeichen dieses Strings in ihrer Reihenfolge den zwölf 8Bit-Vektoren für die Ziffern 0-9 und die beiden Sonderzeichen *'Minus'* und *'Punkt'*.

Auf ähnliche Art und Weise könnte auch ein digital anmutendes Alphabet aufgebaut werden.

Dies war eine - für dieses Buch außergewöhnlich lange - Erklärung zu einem Beispielprogramm, aber ich bin der Meinung, daß gerade kleine Programmiertricks wie diese

näher erläutert werden sollten, um den Inhalt und Ablauf einer solchen Routine deutlich zu machen. Die Könner unter Ihnen wären sicher von allein drauf gekommen, indem sie ganz einfach die *'digit'*-Prozedur in ihre Einzelteile zerlegt hätten, um die Auswirkung jedes einzelnen Befehls separat zu betrachten. Den Anfängern soll hiermit geholfen werden, die - manchmal doch etwas schräge - Programmiererlogik nachzuvollziehen. Wenn Sie sich länger mit der Programmiererei befassen, werden Sie nicht selten überrascht feststellen, daß diese *'schräge'* Logik meist doch sehr konsequent und platzsparend ist.

Eine Fall-Abfrage per **SELECT** oder **IF** und die einzelne Zuordnung der möglichen Bitstellungen zu den jeweiligen Ziffern hätte einige 'zig' Bytes an wertvollem Programmspeicher mehr gekostet. Außerdem wird auch zusätzlich die rationelle Verwendung von Bitvektoren verdeutlicht.

VAL() String => Numerisch

```
Var=VAL(''Text'')
```

Wandelt alle am Anfang des angegebenen ***'Text'***-Strings stehenden Zeichen, die sich zur Darstellung numerischer Werte eignen, in eine dezimale (!) Realzahl um. ***'Text'*** ist eine beliebige Zeichenkette, ein Stringausdruck oder eine String-Variable, deren Inhalt vom Anfang ausgehend daraufhin untersucht wird, ob Textzeichen enthalten sind, die einen Wert in einem der vier Zahlensysteme darstellen. Die Suche wird abgebrochen, wenn das Stringende erreicht ist oder die Funktion auf ein Textzeichen trifft, welches nicht wandelbar ist. Ist das erste Zeichen des Strings ein nicht wandelbares Textzeichen oder ist der String leer, wird eine Null zurückgegeben.

Beispielsweise durch:

```
A$=''1011010"              // <-Binär
PRINT VAL(''&X''+A$)       // Ausgabe: 90

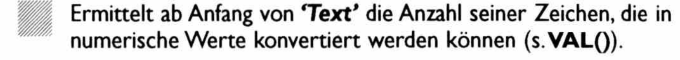
A$=''1141331"              // <-Octal
PRINT VAL(''&O''+A$)       // Ausgabe: 312025

A$=''AF451DE''             // <-Hexadezimal
PRINT VAL(''&H''+A$)       // Ausgabe: 183783902
```

lassen sich auch **HEX$**-, **OCT$**- und **BIN$**-Werte extrahieren. Beachten Sie auch das Beispiel zu **STR$()**.

VAL?() Anzahl wandelbarer Textzeichen ermitteln

```
Var=VAL?(''Text'')
```



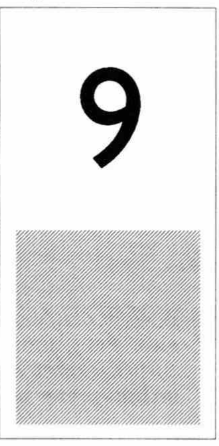
Ermittelt ab Anfang von ***'Text'*** die Anzahl seiner Zeichen, die in numerische Werte konvertiert werden können (s. **VAL()**).

'Text' steht für eine beliebige Zeichenkette oder Stringvariable, die auf die Anzahl ihrer wandelbarer Zeichen untersucht werden soll. Trifft die Funktion auf nicht wandelbare Zeichen, wird die Untersuchung abgebrochen und die Anzahl der bis dahin gefundenen wandelbaren Zeichen geliefert.

Notizen:

Programm-Kontrolle

9



9. PROGRAMM - KONTROLLE

9.1. PROGRAMMSTART UND -ENDE

CHAIN { CHAI } Programm laden (Autostart)

```
CHAIN ''Programmname''
```

Lädt ein BASIC-Programm von Diskette und startet es selbsttätig. Der BASIC-Arbeitsspeicher samt Inhalt (aufrufendes Programm) und die Variablenbelegung wird vorher gelöscht.

CONT { CON } Programm (nach STOP-Befehl) fortsetzen

```
CONT
```

Wurde der Programmlauf mit **STOP** unterbrochen, kann durch CONT im Direktmodus das Programm in der Zeile nach dem **STOP**-Befehl fortgesetzt werden. CONT ist nicht möglich, wenn nach **STOP** entweder **CLEAR** verwendet, das Programmlisting verändert oder neue Variablen eingeführt wurden. Bitte verwechseln Sie diesen Befehl nicht mit der **CONT**-Anweisung in **SELECT..CASE**-Strukturen

EDIT { ED } Programm beenden

```
EDIT
```

Hat dieselbe Wirkung wie **END**. EDIT kehrt jedoch ohne Vorwarnung direkt zum Editor (Interpreter) zurück.

END Programm beenden

```
END
```

Bewirkt den Abbruch des aktuellen Programmlaufes. Es erscheint eine Programm-Ende-Meldung, nach welcher zum Editor zurückgekehrt wird. Variableninhalte und offene Dateien bleiben im Interpreter bis zur nächsten Programmänderung bzw. bis zum

nächsten **CLEAR** erhalten bzw. geöffnet und können im *'Direkt'*-Modus weiter angesprochen werden. Das Programm kann danach nicht durch **CONT** fortgesetzt werden.

QUIT { QU } Programmende (Rückkehr zum DOS)

```
QUIT [x]
```



QUIT ist identisch mit **SYSTEM** und bewirkt, daß das Programm ohne jegliche Sicherheitsabfrage zum DOS-Aufrufer zurückkehrt.

Es kann optional in **'x'** ein 16Bit-Wert angegeben werden, der an das aufrufende Programm zurückgegeben und dann dort ausgewertet werden kann.

Allgemeine Konvention:

x = 0	das Programm wurde ohne Fehler verlassen
x > 0	ein interner BASIC-Fehler ist aufgetreten
x < 0	ein DOS oder BIOS-Fehler ist aufgetreten

In den Fällen, daß **'x'** ungleich Null ist, könnte gfls. `'QUIT ERR'` in einer Fehler-Abfangroutine als Programmende eingesetzt werden.

RUN { RU } Programm starten

```
RUN [''Programmname'']
```

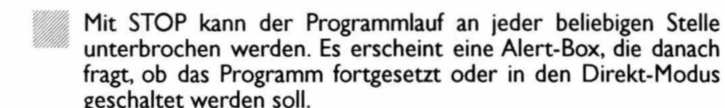


Startet das im Arbeitsspeicher befindliche Programm. Dabei werden sämtliche Variableninhalte gelöscht. RUN kann auch im Direkt-Modus verwendet werden.

Durch die Option ***'Programmname'*** kann ein GFA-BASIC-Programm angegeben werden, das dann in den Arbeitsspeicher geladen und automatisch gestartet wird (vgl. **CHAIN**). Wird bei ***'Programmname'*** keine Extension angegeben, so wird intern automatisch *'.GFA'* eingesetzt.

STOP — Programm unterbrechen

```
STOP
```



Mit STOP kann der Programmlauf an jeder beliebigen Stelle unterbrochen werden. Es erscheint eine Alert-Box, die danach fragt, ob das Programm fortgesetzt oder in den Direkt-Modus geschaltet werden soll.

Es werden keine Variablen gelöscht und auch keine Dateien geschlossen. Im Direktmodus können jetzt beliebige Einzeilenbefehle und -Funktionen ausgeführt werden

z.B.

```
PRINT Var$
```

oder

```
GET x,y,xx,yy,Var$
```

oder

```
Var=xyz
```

Das Programm kann anschließend gfls. durch Eingabe von **CONT** im Direkt-Modus fortgesetzt werden (s. **CONT**).

SYSTEM { SYST } — Programmende (Rückkehr zum DOS)

```
SYSTEM [x]
```

Ist identisch mit **QUIT** (s. Erläuterungen dort).

9.2. LÖSCH - OPERATIONEN

CLEAR { CLE } — Felder und Variablen löschen

```
CLEAR
```

Alle numerischen Variablen erhalten den Wert 0, alle Stringvariablen werden zu Leerstrings (""). Felder werden gelöscht und ihre Dimensionierung aufgehoben. CLEAR darf nicht in **PROCEDURE**'s, **FUNKTION**'s oder **FOR..NEXT**-Schleifen verwendet werden. Bei Programmstart wird CLEAR automatisch ausgeführt.

CLR Einzelvariablen löschen

```
CLR Var [,Var%,Var$,...]
```


Es kann eine Liste von Variablen (keine Feldvariablen) übergeben werden, deren Inhalte dann gelöscht werden.

CLS Bildschirm löschen

```
CLS
```


Löscht den Ausgabe-Bildschirm bzw. das jeweils geöffnete GFA-Window und setzt den Cursor auf *'Home'* (linke, obere Ecke).

NEW Programmspeicher löschen

```
NEW
```


Löscht den BASIC-Arbeitsspeicher mitsamt dem Programm und seinen Variablen. Der Speicher ist dann frei für neue Programme, die z.B. durch die Editorfunktion *'Merge'* geladen werden sollen.

9.3. ZEIT - OPERATIONEN

DATE$ Systemdatum ermitteln

```
Var$=DATE$
```


DATE$ ist eine reservierte Stringvariable. Sie enthält einen String mit dem aktuellen Systemdatum im Format

`'TT.MM.JJJJ'` (bei MODE 0 und MODE 2)

bzw. im US-Format

`'MM/DD/YYYY'` (bei MODE 1 und MODE 3).

DATE$= — Systemdatum bestimmen

```
DATE$=''Datum-String''
```

Mit diesem Befehl ist es möglich, der reservierten Variablen DATE$ einen neuen ***"Datum-String"*** zuzuweisen. Das Format dieses Strings ist von der **MODE**-Einstellung abhängig (s. **MODE**, **SETTIME** und **DATE$**).

DELAY { DEL } — Programm-Unterbrechnung (1 Sekunde)

```
DELAY Sekunden
```

'Sekunden' bestimmt, wieviel Sekunden das Programm pausieren soll.

PAUSE { PA } — Programm-Unterbrechnung (1/50 Sekunde)

```
PAUSE Dauer
```



'Dauer' bestimmt in 50stel Sekunden, wie lange das Programm pausieren soll.

SETTIME { SETT } — Uhrzeit und Datum einstellen

```
SETTIME Zeit$,Datum$
```

In ***'Zeit$'*** und ***'Datum$'*** wird die neue Systemzeit und das neue Systemdatum bestimmt.

Europa-Format:

Zeit$ = `''hh:mm:ss''` oder `''hhmmss''`

USA-Format :

Datum$ = `''mm/dd/yyyy''` oder `''mm/dd/yy''`

Wird das Format (s. **MODE**) nicht korrekt eingehalten, wird SETTIME ignoriert.

TIME$ — System-Uhrzeit ermitteln

```
Var$=TIME$
```


TIME$ ist eine reservierte Stringvariable. Sie enthält einen String mit der aktuellen System-Uhrzeit im Format `'HH.MM.SS'`.

TIME$= — System-Uhrzeit bestimmen

```
TIME$=''Zeit-String''
```


Mit diesem Befehl ist es möglich, der reservierten Variablen TIME$ einen neuen ***"Zeit-String"*** im Format `'hh:mm:ss'` oder `'hhmmss'` zuzuweisen und damit die interne System-Uhr zu 'stellen' (s. auch **SETTIME**).

TIMER — Laufzeit ermitteln

```
Var=TIMER
```


TIMER ist eine reservierte Variable. Sie enthält die seit Systemstart verstrichene Zeit in 1000stel Sekunden.

9.4. FEHLER - BEHANDLUNG

ERR — Fehlercode ermitteln

```
Var=ERR
```


ERR ist eine reservierte Variable, die nach Auftreten eines Fehlers seine Identifikationsnummer enthält.

ERR$() — Fehlertext liefern

```
Var$=ERR$(Index)
```


Die Funktion ERR$() liefert den Text der GFA-BASIC-Fehlermeldung, deren Fehler-***'Index'*** angegeben wurde (s. **ERR** und ANHANG 'FEHLERLISTE').

ERROR { ERR } Fehler simulieren

```
ERROR Fehlernummer
```


'Fehlernummer' steht für die Identifikationsnummer des zu simulierenden Fehlers. Es wird entweder eine Fehlermeldung ausgegeben und das Programm beendet, oder wenn **ON ERROR GOSUB** verwendet wurde, zu der dort angegebenen Prozedur verzweigt.

FATAL Fehlerart ermitteln

```
Var=FATAL
```


FATAL ist eine reservierte Variable. Es wird eine Unterscheidung zwischen *'normalen'*- und *'fatalen'* Fehlern getroffen. *'Fatal'* bedeutet, daß die Adresse des zuletzt ausgeführten BASIC-Befehls nicht mehr bekannt ist. Dies kann passieren, wenn innerhalb einer DOS- oder BIOS-Funktion unabhängig vom BASIC ein Fehler auftritt. Nach Fehlern dieser Art ist die Bearbeitung von **RESUME**-Anweisungen nicht mehr bzw. nur noch **RESUME 'label'** möglich. Ist solch ein *'fataler'* Fehler aufgetreten, so liefert FATAL ein **TRUE** (-1), andernfalls **FALSE** (0).

ON ERROR [GOSUB] Verzweigung bei Fehler

```
ON ERROR [GOSUB] Prozedur
ON ERROR
```


Verzweigt im ersten Fall zur angegebenen ***'Prozedur'***, sobald ein System- oder BASIC-Fehler auftritt. Dazu wird der Name einer ***'Prozedur'*** angegeben, zu welcher das Programm in diesem Fall verzweigen soll (***GOSUB*** kann enfallen). Wurde zu einer Fehler-Routine verzweigt, schaltet der Interpreter die Fehlerbehandlung nach Abarbeiten der angegebenen ***'Prozedur'*** wieder in den Normalmodus zurück. Um dann weiterhin bei Fehlern die Fehlerbehandlung aufrufen zu können, muß vor der **RESUME**-Anweisung zum Verlassen der Behandlungs-'Prozedur' erneut ein ON ERROR GOSUB verfügt werden.

Die zweite Syntax-Variante schaltet den normalen Error-Modus (Programm abbrechen - Fehlermeldung anzeigen) wieder ein.

RESUME { RESU } Programm nach ON ERROR GOSUB fortsetzen

RESUME	->	Fortsetzung mit Wiederholung der fehlerhaften Zeile
RESUME NEXT	->	Fortsetzung mit der Zeile, die der fehlerhaften Zeile folgt
RESUME Label	->	Fortsetzung mit der auf das ´Label´ folgenden Programmzeile

RESUME bestimmt als Abschluß der **ON ERROR GOSUB**-Prozedur, ab welcher Programmzeile das Programm nach Auftreten eines selbstverwalteten Fehlers fortgesetzt werden soll.

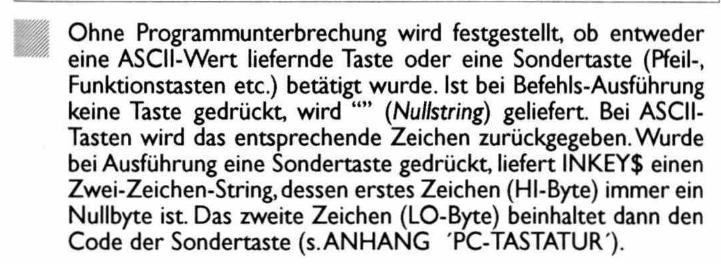
Befindet sich bei der dritten Syntax-Variante das angegebene ***'Label'*** außerhalb der Error-Routine, wird das **GOSUB**-Sprungregister gelöscht und alle globalen Variablen restauriert.

Nach Fatal-Errors (s. **FATAL**) ist ausschließlich ***RESUME 'Label'*** zu verwenden. ***RESUME NEXT*** und ***RESUME*** könnten dann zum Absturz des Rechners und damit zum Totalverlust der Daten führen.

9.5. TASTATUR - KONTROLLE

INKEY$ Einzelzeichen von Tastatur einlesen

Var$=INKEY$	->	direkte Zeichen-Zuweisung
IF LEN(INKEY$)	->	wenn Inkey$ größer '''' ist
IF INKEY$=''Z''	->	wenn '' Z '' gedrückt wurde



Ohne Programmunterbrechung wird festgestellt, ob entweder eine ASCII-Wert liefernde Taste oder eine Sondertaste (Pfeil-, Funktionstasten etc.) betätigt wurde. Ist bei Befehls-Ausführung keine Taste gedrückt, wird "" (*Nullstring*) geliefert. Bei ASCII-Tasten wird das entsprechende Zeichen zurückgegeben. Wurde bei Ausführung eine Sondertaste gedrückt, liefert INKEY$ einen Zwei-Zeichen-String, dessen erstes Zeichen (HI-Byte) immer ein Nullbyte ist. Das zweite Zeichen (LO-Byte) beinhaltet dann den Code der Sondertaste (s. ANHANG ´PC-TASTATUR´).

KEYGET { K } auf Taste warten und Code liefern

```
KEYGET Taste&
```

Es wird auf einen Tastendruck (jedoch nicht Sondertasten wie <NumLock>, <CapsLock>, <Shift>, <Alt> etc.) gewartet und anschließend in der angegebenen numerischen Variablen ***'Taste&'*** einen 16Bit-Wert zurückgegeben.

LO-Byte	(Bits 0 bis 7)	=	ASCII-Code der Taste
HI-Byte	(Bits 8 bis 15)	=	Scan-Code der Taste

Wird eine Byte-Integervariable für ***'Var'*** (Var|) verwendet, wird nur der ASCII-Code im LO-Byte des beschriebenen Word-Wertes geliefert (s. ANHANG 'PC- TASTATUR').

KEYTEST { KEYT } Tastatur durchlaufend abfragen

```
KEYTEST Taste&
```

Es wird in ***'Taste&'*** ein 16Bit-Wert zurückgegeben, welcher der bei Befehlsausführung gedrückten Taste entspricht (s. **KEYGET**). Es wird - wie bei **INKEY$** - nicht auf den Tastendruck gewartet. Wurde keine Taste gedrückt, wird Null geliefert (s.ANHANG 'PC-TASTATUR').

9.6. DEBUGGING

TRACE$ aktuelle Befehlszeile liefern

```
Var$=TRACE$
```

Innerhalb der durch **'TRON Proc'** bestimmten Prozedur kann durch die reservierte Variable TRACE$ der Programmtext der als nächstes abzuarbeitenden Programmzeile ermittelt werden.

TROFF { TROF } Trace-Modus ausschalten

```
TROFF
```



Im Anschluß an TROFF (*'Trace off'*) ist das Programm wieder im normalen Ausführungsmodus. **TRON** oder **'TRON Proc'** sind dann ausgeschaltet.

TRON { TR } Trace-Modus einschalten

```
TRON [#Kanal]
```



TRON (*'Trace on'*) kann an jeder beliebigen Programmstelle eingesetzt werden und gibt ab dann während des Programmlaufs die jeweils aktuelle Programmzeile entweder auf dem Bildschirm oder in die durch ***'#Kanal'*** angegebene und offene Output-Datei (z.B. auch '**COM1:**', '**LPT1:**' oder - für perfektes Debugging - auf dem Zweitbildschirm '**MON:**'; siehe unter **OPEN**).

TRON Proc { TR } Trace-Modus in Prozedur lenken

```
TRON Prozedur
```



Lenkt den Trace-Modus in die angegebene ***'Prozedur'***. Es wird nicht - wie bei **TRON** - automatisch die aktuelle Befehlszeile ausgegeben, sondern es kann beliebig auf die durch **TRACE$** ermittelte, nächste Befehlszeile reagiert werden.

z.B.:

```
TRON Debug_routine
...
... Programm
...
TROFF
'
PROCEDURE Debug_routine
  IF TRIM$(TRACE$)= ...
    ... evtl Verzweigung
  ENDIF
  ...oder
  PRINT TRACE$
  ...oder
  Var$=TRACE$
  ...etc.
RETURN
```

9.7. DIVERSES

DEFNUM { DEF } Stellen-Begrenzung von Ziffern-PRINTs

```
DEFNUM Stelle
```

'Stelle' gibt die Ziffernstelle an, auf die alle - durch **PRINT** etc. - auszugebenden Werte gerundet werden sollen. Die Inhalte von Variablen oder die interne Rechengenaugkeit werden hierdurch nicht beeinträchtigt.

Bei Realzahlen wird der Vorkomma-Anteil, der gfls. hinter ***'Stelle'*** liegt, als Nullen ausgegeben. Liegt ***'Stelle'*** im Nachkommabereich, werden alle dahinterliegenden Nachkommastellen ignoriert bzw. auf die Stelle ***'Stelle'***+1 gerundet. Die Rundung erfolgt mathematisch exakt: *INT(Wert+0.5)*.

FALSE Unwahr-Konstante

```
Var=FALSE
```

FALSE ist eine reservierte Variable, die konstant den *'Un'wert* Null (0) enthält (s. Beispiel zu **XLATE$**).

LET { LE } Daten einer Variablen zuweisen

```
LET Var=Wert
LET Var$=''Text''
```

Es wird ***'Wert'*** bzw. ***'Text'*** der angegebenen Variablen zugewiesen. LET ist ein Überbleibsel aus prähistorischen BASIC-Zeiten und in GFA-BASIC eigentlich überflüssig, da der Befehl mit der normalen Zuweisung `'Var=Wert'` bzw. `'Var$=Text'` identisch ist.

Bei Mehrfachzuweisungen ist LET sogar im Nachteil, da bei jedem LET nur eine Variable zugewiesen werden kann. Seinen Sinn hat dieser Befehl nur in der Einhaltung möglichst hoher Kompatibilität zu anderen BASIC-Dialekten.

MODE { MOD } Zahlen- und Datumsformat bestimmen

```
MODE Modus
```



In der Grundeinstellung werden in GFA-BASIC durch **PRINT USING** alle Werte im Europa-Format ausgegeben. Das heißt gfls, daß als Tausendertrennung ein Komma und als Dezimaltrennung ein Punkt verwendet wird. In den USA gelten die umgekehrten Konventionen. Ähnliches gilt für die Darstellung des Datums (s. **SETTIME**).

MODE bestimmt durch den Parameter ***'Modus'*** über die Darstellung des Datums durch **DATE$** und **FILES** bzw. über das Eingabeformat des Datums bei **SETTIME** und **DATE$=**, sowie auch die zu verwendende Art der Werte-Darstellungsart bei **PRINT USING**-Ausgaben bzw. bei Verwendung von **STR$()**.

'Modus':	**USING:**	**DATE$:**
0	#,###.##	30.10.1991
1	#,###.##	30/10/1991
2	#.###,##	30.10.1991
3	#.###,##	30/10/1991

SOUND { SO } Klangausgabe

```
SOUND Hertz,Dauer
```



'Hertz' bestimmt die Tonfrequenz der Klangausgabe. ***'Dauer'*** bestimmt die Zeit (in 1/18 Sekunden), die der Ton bis zur Ausführung des nächsten Befehls gehalten wird (z.B. 2 Sekunden lang Kammerton A: `SOUND 440,36`).

TRUE Wahr-Konstante

```
Var=TRUE
```



TRUE ist eine reservierte Variable, die konstant den *'Wahr'*wert -1 enthält (s. Beispiel zu **XLATE$** und **VAR**).

VOID {VO oder ~} Dummy-Zuweisung

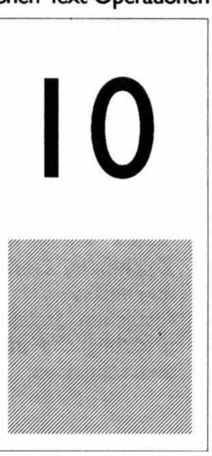
```
VOID Funktion()
~Funktion()
```



VOID ersetzt eine *Dummy-Zuweisung* ('`Dummy=Funktion()`'), ist aber schneller. Es wird eine Funktion aufgerufen, ohne dieser eine Rückgabevariable zur Verfügung zu stellen. VOID kann dann eingesetzt werden, wenn der Rückgabewert einer Funktion uninteressant ist (z.B. `VOID FRE(0)` zum Auslösen einer *'Garbage Collection'* oder `~FSETDTA()` etc.).

Notizen:

Text-Operationen



10. TEXT - OPERATIONEN

10.1. STRING - MANIPULATIONEN

MID$()= Teilstring zuweisen

```
MID$(Text$,Start [,Anzahl])=Quell$
```



'Text$'*** gibt eine Stringvariable an, in welche ***'Quell$' ab der Position ***'Start'*** eingesetzt werden soll.

'Anzahl' gibt optional die Anzahl der Zeichen von ***'Quell$'*** an, die maximal in ***'Text$'*** eingesetzt werden sollen. Fehlt der Parameter ***'Anzahl'***, so werden maximal nur soviele Zeichen von ***'Quell$'*** eingefügt, wie ab ***'Start'*** in ***'Text$'*** noch hineinpassen. Die ursprüngliche Länge von ***'Text$'*** bleibt dabei auf jeden Fall unverändert.

MIRROR$() Zeichenkette spiegeln

```
Var$=MIRROR$(Quell$)
```



Mit ***'Quell$'*** wird ein String angegeben, der dann gespiegelt (z.B. *tlegeipseg*) zurückgeliefert wird. In Verbindung mit **RINSTR()** ist diese Funktion z.B. bei der Pfadanalyse sehr nützlich.

LSET { LS } String in String linksbündig einsetzen

```
LSET Ziel$=Quell$
```



'Quell$'*** kann eine beliebige Zeichenkette oder Stringvariable sein, deren Inhalt linksbündig in die Stringvariable ***'Ziel$' eingefügt wird.

Dabei bleibt die ursprüngliche Länge von ***'Ziel$'*** unverändert. Ist ***'Ziel$'*** kürzer als ***'Quell$'***, werden von ***'Quell$'*** nur soviel Zeichen in ***'Ziel$'*** eingefügt, wie bis zum Stringende von ***'Ziel$'*** hineinpassen. Ist ***'Ziel'*** dagegen länger als ***'Quell$'***, werden die restlichen Stellen von ***'Ziel$'*** mit Leerzeichen aufgefüllt. Der ursprüngliche Inhalt von ***'Ziel$'*** wird in jedem Fall komplett überschrieben.

RSET { RS } String in String rechtsbündig einsetzen

```
RSET Ziel$=Quell$
```



'Quell' kann eine beliebige Zeichenkette oder Stringvariable sein, deren Inhalt rechtsbündig in ***'Ziel$'*** eingefügt wird. Weitere Erläuterungen finden Sie unter **LSET**.

TRIM$() Leerzeichen im String löschen

```
Var$=TRIM$(Quell$)
```



Löscht alle Leerzeichen, die am Anfang oder Ende von ***'Quell$'*** stehen und gibt den verbleibenden Stringteil zurück. ***'Quell$'*** bleibt dabei in seiner Ausgangsform erhalten.

10.2. STRING - ANALYSE

INSTR() String im String suchen

```
Var=INSTR(Text$,Such$ [,Start] )
Var=INSTR( [Start,] Text$,Such$)
```



Liefert einen Wert, der die absolute Position von ***'Such$'*** in ***'Text$'*** angibt. Ist ***'Such$'*** in ***'Text$'*** nicht enthalten oder sind beide Strings leer, so wird der Wert 0 zurückgegeben.

Wird optional in ***'Start'*** eine Startposition angegeben, wird erst ab der damit angegebenen Zeichenposition (inclusiv) gesucht.

LEFT$() Linksbündigen Teilstring ermitteln

```
Var$=LEFT$(Quell$ [,Anzahl])
```



Liefert das erste Zeichen des Strings ***'Quell$'*** bzw. - bei Verwendung des optionalen Parameters ***'Anzahl'*** - soviele Zeichen ab Stringanfang von ***'Quell$'***, wie in ***'Anzahl'*** angegeben wurden. Ist ***'Quell$'*** ohne Inhalt (""), wird auch ein Nullstring zurückgegeben.

MID$() beliebigen Teilstring ermitteln

```
Var$=MID$(Quell$,Start [,Anzahl])
```



Es wird ein Teilstring von ***'Quell$'*** geliefert. ***'Start'*** gibt die Position in ***'Quell$'*** an, ab welcher gelesen werden soll.

Wird die Option ***'Anzahl'*** verwendet, werden ab ***'Start'*** soviel Zeichen geliefert, wie in ***'Anzahl'*** angegeben sind. Sonst wird ab ***'Start'*** der gesamte Rest von ***'Quell$'*** zurückgegeben. Ist ***'Quell$'*** ohne Inhalt, wird eine Nullstring ("") geliefert.

Beachten Sie auch das Beispiel zu **STR$()**.

RIGHT$() Rechtsbündigen Teilstring ermitteln

```
Var$=RIGHT$(Quell$ [,Anzahl])
```



Liefert das letzte Zeichen des Strings ***'Quell$'*** bzw. es werden bei Verwendung des optionalen Parameters ***'Anzahl'*** - ab Stringende rückwärts gezählt - soviele Zeichen von ***'Quell$'*** geliefert, wie in ***'Anzahl'*** angegeben wurden. Ist ***'Quell$'*** ohne Inhalt (""), wird auch ein Nullstring zurückgegeben.

RINSTR() String im String rückwärts suchen

```
Var=RINSTR(Text$,Such$ [,Start] )
Var=RINSTR([Start,] Text$,Such$)
```



Liefert einen Wert, der die Position von ***'Such$'*** in ***'Text$'*** angibt. Bei der Durchsuchung des Strings wird am Stringende begonnen. Weiteres siehe **INSTR**.

10.3. STRING-ARITHMETIK

LEN() — Type-/Stringlänge ermitteln

```
Var=LEN(Var$)
Var=LEN(Typenname:)
Var=LEN(Variablenname.)
```

Ermittelt entweder die Länge des angegebenen Strings ***'Var$'*** (Syntax-Variante 1) *oder* die Gesamtlänge eines **TYPE**-Blocks (Variante 2) *oder* die Länge einer **TYPE**-Variablen (Variante 3).

MAX($) — Größten String ermitteln

```
Var$=MAX(Expr1$,Expr2$ [,Expr3$,...])
```

Es wird der größte String einer Stringliste ermittelt, indem der Reihe nach alle Einzelzeichen der zu vergleichenden Strings überprüft werden. Haben die jeweiligen Zeichen denselben ASCII-Wert, werden solange die jeweils nächsten Zeichen geprüft, bis Sie sich voneinander unterscheiden oder einer der verglichenen Strings keine Zeichen mehr enthält. Im ersten Fall ist der Ausdruck größer, dessen zuletzt geprüftes Zeichen den größeren ASCII-Wert besitzt. Im zweiten Falle ist es der String mit der größeren Länge.

'Expr$' kann eine beliebiger Textausdruck, ein String oder eine Stringvariable sein.

MIN($) — Kleinsten String ermitteln

```
Var$=MIN(Expr1$,Expr2$ [,Expr3$,...])
```

Gibt den kleinsten String einer Stringliste zurück. Weitere Erläuterungen finden Sie analog unter **MAX($)**.

PRED($) nächstkleineres ASCII-Zeichen ermitteln

```
Var$=PRED(Expr$)
```

Liefert das nächstkleinere ASCII-Zeichen des ersten Zeichens von ***'Expr$'***, indem dessen ASCII-Wert um 1 vermindert wird.

SUCC($) nächstgrößeres ASCII-Zeichen ermitteln

```
Var$=SUCC(Expr$)
```

Liefert das nächstgrößere ASCII-Zeichen des ersten Zeichens von ***'Expr$'***, indem dessen ASCII-Wert um 1 erhöht wird.

10.4. STRING - FORMATIERUNG

SPACE$() Leerzeichen-String bilden

```
Var$=SPACE$(Anzahl)
```



Es wird ein String aus ***'Anzahl'*** Leerzeichen gebildet.

STRING$() Mehrfach-Zeichenkette bilden

```
Var$=STRING$(Anzahl,''Text'')
 Var$=STRING$(Anzahl,Ascii)
```



Es wird ein String gebildet, der sich daraus ergibt, daß ***'Text'*** so oft verkettet wird, wie in ***'Anzahl'*** angegeben. ***'Text'*** kann auch als Variable oder Stringausdruck angegeben werden.

Soll ein einzelnes Zeichen multipliziert werden, kann statt ***'Text'*** auch der ASCII-Wert des Zeichens verwendet werden. Die entstehende Zeichenkette darf nicht mehr als 32767 Zeichen enthalten (maximale Größe einer Stringvariablen).

10.5. STRING - UMWANDLUNG

LCASE$() PC-spezifische Umwandlung groß => klein

```
Var$=LCASE$(Quell$)
```

Trifft LCASE$() beim Lesen von ***'Quell$'*** auf einen großgeschriebenen Buchstaben (ASCII-Werte 65 bis 90), so wird dieser in den entsprechenden Kleinbuchstaben (ASCII-Werte 97 bis 129) umgewandelt. Alle anderen Zeichen (auch Umlaute) bleiben unverändert.

LOWER$() Buchstabenumwandlung groß => klein

```
Var$=LOWER$(Quell$)
```

Es gelten die gleichen Erläuterungen wie zu **LCASE$()**, nur daß LOWER$() zusätzlich auch großgeschriebene PC-Umlaute (Ä, Ö, Ü) in Kleinbuchstaben (ä, ö, ü) wandelt.

UCASE$() PC-spezifische Umwandlung klein => groß

```
Var$=UCASE$(Quell$)
```

UCASE$() bildet die exakte Umkehrung der Funktion **LCASE$()**. Weitere Erläuterungen finden Sie dort.

UPPER$() Buchstabenumwandlung klein => groß

```
Var$=UPPER$(Quell$)
```

UPPER$() bildet die exakte Umkehrung der Funktion **LOWER$()**. Weitere Erläuterungen finden Sie dort.

XLATE$() Buchstabenumwandlung nach freier Tabelle

```
Var$=XLATE$(Quell$,Feld|())
```

Diese Stringfunktion wandelt den vorgegebenen String ***'Quell$'*** anhand einer frei definierbaren Wertetabelle ***'Feld|()'*** in einen

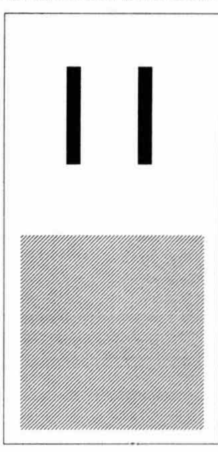
anderen String um. ***'Feld|()'*** ist ein Bytefeld mit 256 Elementen, die jeweils den entsprechenden ASCII-Wert darstellen. In diese Tabelle können nun nach eigenem Ermessen ASCII-Werte eingetragen werden, in die dann die einzelnen Zeichen des Ursprungsstrings ***'Quell$'*** umgewandelt werden.

z.B.:

```
a$=''GFA-BASIC ist super!!!''// Arbeitsstring
OPEN ''o'',#1,''vid:''       // Video-Kanal öffnen
DIM ct|(255)                 // XLATE-Code-Tabelle
DIM dt|(255)                 // XLATE-Decode-Tabelle
DIM kt!(255)                 // Kontrolltabelle
FOR i&=0 TO 255              // alle ASCII's...
  ct|(i&)=i&                 // Tabelle 1 belegen
  dt|(i&)=i&                 // Tabelle 2 belegen
NEXT i&
FOR i&=0 TO 127              // halbe ASCII-Tabelle
  REPEAT
    m|=RAND(128)             // ein Zeichen aus der
    '                        // 1. ASCII-Hälfte
  UNTIL kt!(m|)=FALSE        // ASCII-Position frei?
  REPEAT
    n|=255-RAND(128)         // ein Zeichen aus der
    '                        // 2. ASCII-Hälfte
  UNTIL kt!(n|)=FALSE        // ASCII-Position frei?
  kt!(m|)=TRUE,kt!(n|)=TRUE // Kontrollpos. belegen
  SWAP ct|(m|),ct|(n|)       // Zeichen vertauschen
  dt|(m|)=n|,dt|(n|)=m|      // Decode-Tab. tauschen
NEXT i&
b$=XLATE$(a$,ct|())          // Verschlüsselung
PRINT ''TEXT  : '';a$;CHR$(10)// Orig.-Text ausgeben
PRINT #1,''CODE  : '';b$;    // Code-Text ausgeben
PRINT CHR$(10);''DECODE: '';
PRINT XLATE$(b$,dt|())       // decodieren u.ausgeben
CLOSE #1                     // 'VID:' schließen
```

Notizen:

Arithmetik



11. ARITHMETIK

11.1. ARITHMETISCHE OPERATOREN

+	Additionsoperator
-	Subtraktionsoperator
*	Multiplikationsoperator
/	Divisionsoperator
^	Potenzierungsoperator
=	Zuweisungsoperator
MOD	Modulo-Operator

z.B.

```
a%=113^(((34+6)*2-50/2) MOD 5)    (ergibt 1)
```

11.2. OPERATOREN INCL. VARIABLENZUWEISUNG

+=	Variablenzuweisung incl. Addition
-=	Variablenzuweisung incl. Subtraktion
*=	Variablenzuweisung incl. Multiplikation
/=	Variablenzuweisung incl. Division
%=	Variablenzuweisung incl. Modulo-Berechnung
&=	Variablenzuweisung mit Bit-Konjunktion
|=	Variablenzuweisung mit inclusiver Bit-Disjunktion
^=	Variablenzuweisung mit exclusiver Bit-Disjunktion

z.B.

```
a%=16,b%=4      // weise a% den Wert 16
                // und b% den Wert 4 zu
a%-=b%          // subtrah. b% von a%
PRINT a%        // gibt 12 aus
b%*=a%          // multipl. b% mit a%
PRINT b%        // gibt 48 aus
a%+=8           // addiere 8 zu a%
PRINT a%        // gibt 20 aus
b% %= a%        // Modulo von b% durch a%
PRINT b%        // gibt 8 aus
```

11.3. VERGLEICHSOPERATOREN

=		*'gleich'*-Operator
>		*'größer als'*-Operator
<		*'kleiner als'*-Operator
<> oder ><		*'ungleich'*-Operator
>= oder =>		*'größer gleich'*-Operator
<= oder =<		*'kleiner gleich'*-Operator
==		*'ungefähr gleich'*-Operator

z.B.

```
IF (a%=>b% OR c%=d%) AND e%<>f%
```

11.4. LOGISCHE OPERATOREN

AND oder && — Konjunktion zweier Wahrheitswerte

```
Arg1 AND Arg2
Arg1 && Arg2
```

Das Ergebnis von AND ist nur dann *'wahr'* (**TRUE**=-1), wenn beide Argumente *'wahr'* sind.

z.B.:

```
IF (2+4=6) AND (15/5=3)
  PRINT ''es ist beides wahr''
ENDIF
```

EQV — Äquivalenz zweier Wahrheitswerte

```
Arg1 EQV Arg2
```

Umkehrung zu **XOR**. Das Ergebnis ist dann logisch *'wahr'*, wenn die beiden Argumente entweder beide *'wahr'* oder beide *'unwahr'* sind. Ist nur ein Argument *'wahr'* und das andere nicht, wird generell *'unwahr'* festgestellt.

z.B.:

```
IF (2+4=19) EQV 111*0
  PRINT ''beide sind unwahr, also wahr''
ENDIF
```

IMP — Implikation zweier Wahrheitswerte

```
Arg1 IMP Arg2
```



Eine Implikation ist nur dann *'unwahr'* (**FALSE**=0), wenn aus etwas Wahrem etwas Falsches folgt. D.h. das Ergebnis ist immer dann *'unwahr'*, wenn ***'Arg1'*** *'wahr'* und ***'Arg2'*** *'unwahr'* ist. In allen anderen Fällen ist das Ergebnis *'wahr'*.

NOT oder ! — Negation eines Wahrheitswertes

```
NOT Arg ! Arg
```



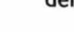
Vertauscht Wahrheitswerte ins Gegenteil.

z.B.:

```
IF NOT TRUE=FALSE
  PRINT ''nicht wahr ist auch unwahr''
ENDIF
```

OR oder || — incl. Disjunktion zweier Wahrheitswerte

```
Arg1 OR Arg2
Arg1 || Arg2
```



Das Ergebnis von OR ist schon dann logisch *'wahr'*, wenn nur eines der beiden Argumente *'wahr'* ist.

z.B.:

```
IF (2+4=23) OR (15/5=3)
  PRINT ''eins von beiden ist wahr''
ENDIF
```

XOR oder ^^ — excl. Disjunktion zweier Wahrheitswerte

```
Arg1 XOR Arg2
Arg1 ^^ Arg2
```



Ausschließendes Oder. Das Ergebnis von XOR ist dann logisch *'wahr'* (**TRUE**=-1), wenn nur eines der beiden Argumente *'wahr'* ist. Sind beide oder keins *'wahr'*, wird *'unwahr'* (**FALSE**=0) festgestellt.

z.B.:

```
IF (2+4=6) XOR (15/5=3)
  PRINT ''beide sind wahr, also unwahr''
ENDIF
```

11.5. OPERATOREN - PRIORITÄT

```
-------------------------------------------------höchste Priorität
      ( )          Klammern
       +           Stringaddition

      =  <>        |
      >  <         |- Stringvergleichsoperatoren
      >=  <=       |

      +    -       Vorzeichen
      ^            Potenzierung
      *  /         Multiplikation und Division
      DIV MOD      Ganzzahldivision und Modulo
      +  -         Addition und Subtraktion

      =  <>        |
      >  <         |- num. Vergleichsoperatoren
      >=  <=       |

      AND  OR      |
      XOR          |- bit-arithmetische Operatoren
      IMP EQV      |

      AND  OR      |
      XOR          |- logische Operatoren
      IMP EQV      |

      NOT          bit-arithmetische Negation
      NOT          logische Negation
-----------------------------------------------geringste Priorität
```

11.6. MATHEMATISCHE GRUNDFUNKTIONEN

DEC / -- Integer-Dekrementierung um 1

```
DEC Var
Var--
```

Vermindert den Wert von ***'Var'*** um 1.

INC / ++ Integer-Inkrementierung um 1

```
INC Var
Var++
```

Erhöht den Wert von ***'Var'*** um 1.

ADD { AD } Additionsbefehl

```
ADD Var,Wert
```

Addiert ***'Wert'*** zu ***'Var'*** und legt das Ergebnis in ***'Var'*** ab.

ADD() Integer-Additionsfunktion

```
Var=ADD(Wert1,Wert2)
```

Addiert ***'Wert1'*** zu ***'Wert2'*** und liefert das entsprechende Integer-Ergebnis.

Anmerkung:

Werte-Parameter können grundsätzlich in GFA-BASIC als Konstante, Variable, Ausdruck oder Funktion angegeben werden. Bei Integer-Operationen - wie z.B. ADD() - werden gfls. Realwerte vorher intern auf ihren Integeranteil reduziert. Die Inhalte übergebener Variablen-Parameter bleiben dabei jedoch unverändert (Ausnahmen: z.B. '**Var=+1**'*).*

DIV — Divisionsbefehl

```
DIV Var,Wert
```



Dividiert *'Var'* durch *'Wert'* und legt das Ergebnis in *'Var'* ab. Wird in *'Var'* eine Integer-Variable angegeben, wird ein evtl. in *'Wert'* angegebener Realwert als Integerwert behandelt.

DIV() — Integer-Divisionsfunktion

```
Var=DIV(Wert1,Wert2)
```



Dividiert *'Wert1'* durch *'Wert2'* und liefert das entsprechende Integer-Ergebnis. Siehe Anmerkung zu **ADD()**.

MUL { MU } — Multiplikationsbefehl

```
MUL Var,Wert
```



Multipliziert *'Var'* mit *'Wert'* und legt das Ergebnis in *'Var'* ab. Wird in *'Var'* eine Integer-Variable angegeben, wird ein evtl. in *'Wert'* angegebener Realwert als Integerwert behandelt.

MUL() — Integer-Multiplikationsfunktion

```
Var=MUL(Wert1,Wert2)
```



Multipliziert *'Wert1'* mit *'Wert2'* und liefert das entsprechende Integer-Ergebnis. Siehe Anmerkung zu **ADD()**.

SUB — Subtraktionsbefehl

```
SUB Var,Wert
```



Subtrahiert *'Wert'* von *'Var'* und legt das Ergebnis in *'Var'* ab.

SUB() Integer-Subtraktionsfunktion

```
Var=SUB(Wert1,Wert2)
```


Subtrahiert ***'Wert2'*** von ***'Wert1'*** und liefert das entsprechende Integer-Ergebnis. Siehe Anmerkung zu **ADD()**.

11.7. SPEZIELLE ARITHMETIK

ABS() Absolut-Betrag ermitteln

```
Var=ABS(Arg)
```


ABS() gibt das Argument ***'Arg'*** vorzeichenlos (*absolut*) als positiven Wert zurück.

CFLOAT() Integerwert in Fließkommawert wandeln

```
Var=CFLOAT(Arg)
```


Wandelt ***'Arg'*** (*Integerwert*) in eine Realzahl um. CFLOAT() bildet die Umkehrfunktion zu **CINT()**.

CINT() Fließkommawert in Integerwert wandeln

```
Var=CINT(Arg)
```


Wandelt ***'Arg'*** (*Realwert*) in eine Integerzahl um. Im Gegensatz zu **INT()** wird die Zahl vorher exakt gerundet. CINT() bildet die Umkehrfunktion zu **CFLOAT()**.

EVEN() Zahl auf 'gerade' testen

```
Var=EVEN(Arg)
```


Liefert -1 (**TRUE**), wenn ***'Arg'*** gerade ist. Sonst wird 0 (**FALSE**) geliefert.

FRAC() Nachkommastellen ermitteln

```
Var=FRAC(Arg)
```



Liefert den Dezimalanteil (sprich: die *Nachkommastellen*) von ***'Arg'***, falls dies ein Realwert ist. Ist ***'Arg'*** ein Integerwert, wird 0 geliefert.

MOD() Integer-Modula-Funktion

```
Var=MOD(Wert1,Wert2)
```



Berechnet den ganzzahligen Rest der Integer-Divison ***'Wert1'*** durch ***'Wert2'*** (*Modulo*) und liefert das entsprechende Integer-Ergebnis. Siehe Anmerkung zu **ADD()**.

ODD() Zahl auf 'ungerade' testen

```
Var=ODD(Arg)
```



Liefert -1 (**TRUE**), wenn ***'Arg'*** ungerade ist. Sonst wird 0 (**FALSE**) geliefert.

SGN() Vorzeichen ermitteln

```
Var=SGN(Arg)
```



Liefert das Vorzeichen von ***'Arg'***:

1	wenn ***'Arg'*** größer	0
-1	wenn ***'Arg'*** kleiner	0
0	wenn ***'Arg'*** gleich	0

11.8. RUNDUNGSFUNKTIONEN

CEIL() auf nächstgrößere Ganzzahl aufrunden

```
Var=CEIL(Arg)
```

Diese Funktion entspricht bei Realwerten mit Nachkommaanteil INT()+1. Bei Integerwerten ist sie mit **INT()** identisch. Siehe Erläuterungen zu **INT()**.

FIX() vorzeichen-unabhängig auf Ganzzahl runden

```
Var=FIX(Arg)
```

Liefert den ganzzahligen Anteil (Integeranteil) der Real-Zahl ***'Arg'***. Die Funktion rundet Zahlen weder auf noch ab, sondern entfernt nur die Nachkommastellen. Im Minusbereich wird dadurch der Minuswert kleiner.

z.B.:

```
FIX(-12.33)  ergibt  -12
FIX(33.17)   ergibt   33
```

FIX() ist identisch mit **TRUNC()**.

FLOOR() vorzeichen-abhängig auf Ganzzahl abrunden

```
Var=INT(Arg)
```

Wandelt die Real-Zahl ***'Arg'*** in eine Integer-Zahl. Es wird die nächstkleinere Ganzzahl zurückgegeben. Im Minusbereich wird dadurch der Minuswert größer.

z.B.:

```
INT(-12.33)  ergibt  -13
INT(33.17)   ergibt   33
```

FLOOR() ist identisch mit **INT()**.

INT() vorzeichen-abhängig auf Ganzzahl abrunden

`Var=INT(Arg)`



INT() ist identisch mit **FLOOR()**. Weitere Erläuterungen finden Sie dort.

PRED() nächstkleinere Ganzzahl ermitteln

`Var=PRED(Arg)`

Liefert die nächstkleinere Integerzahl vor ***'Arg'***. Bei ***'Arg'*** als Realzahl werden die Nachkommastellen intern vor der Funktionsausführung integriert (also: `Var=PRED(INT(Arg))`).

ROUND() Rundungs-Funktion

`Var=ROUND(Arg [,Stelle] )`



Rundet ***'Arg'*** mathematisch exakt auf eine ganze Zahl. Ist der Nachkomma-Anteil von ***'Arg'*** kleiner als 0.5, so wird zur kleineren Ganzzahl gerundet. Sonst zur nächst größeren. Wird der optionale Parameter ***'Stelle'*** verwendet, wird auf die angegebene Nachkommastelle gerundet. Ist ***'Stelle'*** negativ, wird auf die entsprechende ***'Stelle'*** vor dem Komma gerundet.

SUCC() nächstgrößere Ganzzahl ermitteln

`Var=SUCC(Arg)`

Liefert die nächstgrößere Integerzahl nach ***'Arg'***. Bei ***'Arg'*** als Realzahl werden die Nachkommastellen intern vor der Funktionsausführung integriert (also: `Var=SUCC(INT(Arg))`).

TRUNC() vorzeichen-unabhängig auf Ganzzahl runden

`Var=TRUNC(Arg)`



TRUNC() ist identisch mit **FIX()**. Erläuterungen finden Sie dort.

11.9. ALGEBRAISCHE FUNKTIONEN

EXP() Exponentialfunktion

```
Var=EXP(Arg)
```

Berechnet das Ergebnis des Exponenten ***'Arg'*** zur Basis 'e' (*Euler'sche Zahl*=2.718...).

Gleichbedeutend mit der Potenz:

2.718281828462 ^ ***'Arg'***

EXP() ist die Umkehrfunktion zu **LOG()**.

LOG() natürlicher Logarithmus

```
Var=LOG(Arg)
```



Liefert den entsprechenden Exponenten zur Basis 'e' (*Euler'sche Zahl*: 2.718281828462).

Die Potenz dieser Basis mit diesem Exponenten ergibt dann rückschließend ***'Arg'***:

zu LOG():	***'Arg'*** = e ^ LOG(***'Arg'***)
zu LOG2():	***'Arg'*** = 2 ^ LOG2(***'Arg'***)
zu LOG10():	***'Arg'*** = 10 ^ LOG10(***'Arg'***)

LOG() ist die Umkehrfunktion zu **EXP()**.

LOG2() binärer Logarithmus

```
Var=LOG2(Arg)
```



Liefert den entsprechenden Exponenten zur Basis 2 (lat.: *bi*). Weitere Erläuterung siehe bei **LOG()**

LOG10() dekadischer Logarithmus

`Var=LOG(Arg)`

Liefert den entsprechenden Exponenten zur Basis 10 (griech.: *deka*). Weitere Erläuterung siehe bei **LOG()**

SCALE() Skalierungsfunktion

`Var=SCALE(Faktor,Multiplikator,Divisor)`

Das Integer-Word ***'Faktor'*** wird mit dem Integer-Word ***'Multiplikator'*** multipliziert und das daraus errechnete 32Bit-Ergebnis mit dem Integer-Word ***'Divisor'*** dividiert. Das Endergebnis wird dann wiederum als Integer-Word geliefert.

Bei der Umrechnung von z.B. auflösungsabhängigen Bildschirmkoordinaten leistet diese Funktion wertvolle Dienste. Sie arbeitet gegenüber `'Arg1*Arg2/Arg3'` geschwindigkeitsoptimiert.

SQR() Wurzelfunktion

`Var=SQR(Arg)`

Es wird die 2.Wurzel von ***'Arg'*** geliefert. Wird eine höhere Wurzel gebraucht, so ist diese über den Umweg der Potenzierung mit gebrochenem Exponenten zu berechnen:

3.Wurzel aus ***'Arg'***: ***'Arg'*** ^ (1 / 3)
4.Wurzel aus ***'Arg'***: ***'Arg'*** ^ (1 / 4)
...etc.

11.10. KOMBINATIONSFUNKTIONEN

COMBIN() Binominal-Koeffizienten ermitteln

`Var=COMBIN(Menge,Teilmenge)`

Diese interessante mathematische Funktion berechnet die Anzahl der Möglichkeiten, die sich ergeben, wenn man aus der Grundmenge ***'Menge'*** eine ***'Teilmenge'*** *ziehen* will (z.B. Lotto -> `COMBIN(49,6)`). Dabei wird davon ausgegangen, daß die einmal

gezogenen Elemente nicht wieder zur Verfügung stehen. Es ist also ausgeschlossen, daß sich eine *'COMBINation'* wiederholen kann.

'Teilmenge' kann dabei logisch nur im Bereich 0 bis ***'Menge'*** liegen.

`COMBIN(m,t)` **entspricht:**

```
VARIAT(m,t)/FACT(t)
```

oder auch

```
FACT(m)/(FACT(t)*FACT(m-t))
```

FACT() Fakultätsfunktion

`Var=FACT(Menge)`

FACT() liefert die Fakultät der Ganzzahl ***'Menge'***. Die Fakultät einer ***'Menge'*** ist definiert als das Produkt aller Ganzzahlen dieser ***'Menge'***.

z.B.: `FACT(7)` **entspricht 1*2*3*4*5*6*7 = 5040**

PERMUT() Permutationsfunktion (Variation)

`Var=PERMUT(Menge,Teilmenge)`

Unter einer *'PERMUTation'* ist die Berechnung aller möglichen Kombinationen einer ***'Teilmenge'*** von Elementen innerhalb einer ***'Menge'*** von Elementen. ***'Teilmenge'*** kann dabei logisch nur im Bereich 0 bis ***'Menge'*** liegen.

Im Gegensatz zu **COMBIN()** wird davon ausgegangen, daß nach jeder *Ziehung* die *gezogenen* Elemente wieder zur Verfügung stehen.

z.B.:

Eine Wohngemeinschafts-***'Menge'*** von 5 Bewohnern will ermitteln, wieviele Kombinationen möglich sind, wenn immer jeweils eine 'Teilmenge' von zwei Bewohnern zusammen die wöchentliche Pflege des gemeinsamen Gartens (zwei Beete) übernehmen und dabei aber jeder Bewohner, der schon einmal 'dran' war, wieder zur Kombination mit den anderen Bewohnern für eine weitere ***'Teilmenge'*** zur Verfügung steht. Außerdem soll gegeben sein, daß jeder der beiden Bewohner jeweils nur für eines der beiden Beete zuständig ist und nach jedem Durchgang

jeder Bewohner mindestens einmal an jedem der beiden Beete beschäftigt war.

PERMUT(5,2) ergibt dann 20. Das heißt, daß erst nach zwanzig Wochen eine identische Kombination auftritt. Also eine Paarung wieder aufeinander trifft, von der beide wieder an denselben Beeten wie vor 20 Wochen beschäftigt sind.

PERMUT(m,t) ist identisch mit einem auf *'t'* Multiplikationen beschränkten **FACT(m)**.

z.B. PERMUT(9,6): 9*8*7*6*5*4 = 60480

PERMUT(m,t) ist auch idenstisch mit der Division der Fakultät von *'m'* durch die Fakultät der Differenz aus *'m'* und *'t'*.

z.B. FACT(9)/FACT(9-6):

(9*8*7*6*5*4*3*2*1) DIV (3*2*1) = 60480

VARIAT() Permutationsfunktion (Variation)

```
Var=VARIAT(Menge,Teilmenge)
```



VARIAT() ist identisch mit **PERMUT()**. Erläuterungen finden Sie dort.

11.11. VERGLEICHS-OPERATIONEN

IMAX() Größten Integer-Wert ermitteln

```
Var=IMAX(Expr1,Expr2 [,Expr3,...])
```



Gibt den größten Wert einer Werteliste im Integer-Format zurück. *'Expr'* kann eine beliebiger numerischer Ausdruck, ein Wert oder eine numerische Variable sein.

IMIN() Kleinsten Integer-Wert ermitteln

```
Var=IMIN(Expr1,Expr2 [,Expr3,...])
```

Gibt den kleinsten Wert einer Werteliste im Integer-Format zurück. ***'Expr'*** kann eine beliebiger numerischer Ausdruck, ein Wert oder eine numerische Variable sein.

MAX() Größten Realwert ermitteln.

```
Var=MAX(Expr1,Expr2 [,Expr3,...])
```

Gibt den größten Wert einer Werteliste im Realformat zurück. ***'Expr'*** kann eine beliebiger numerischer Ausdruck, ein Wert oder eine numerische Variable sein.

MIN() Kleinsten Realwert ermitteln

```
Var=MIN(Expr1,Expr2 [,Expr3,...])
```

Gibt den kleinsten Wert einer Werteliste im Realformat zurück. ***'Expr'*** kann eine beliebiger numerischer Ausdruck, ein Wert oder eine numerische Variable sein.

11.12. BEREICHSÜBERPRÜFUNG

BOUND() Prüfung auf Bereichsüberschreitung

```
Var=BOUND(Wert,Minimum,Maximum)
```

Diese Funktion überprüft, ob sich ***'Wert'*** innerhalb des Bereichs zwischen ***'Minimum'*** und ***'Maximum'*** (beides inclusive) befindet. Falls ja, wird ***'Wert'*** unverändert wieder zurückgegeben. Im negativen Fall wird die Fehlermeldung *'Bereichsfehler'* ausgelöst.

Wer diese Fehlermeldung umgehen will, wird die folgende kleine Funktion bemühen müßen. Sie liefert entweder den Wert 0 (**FALSE**), wenn ***'Wert'*** außerhalb, bzw. -1 (**TRUE**), wenn ***'Wert'*** innerhalb des gültigen Wertebereichs (beides inclusiv) liegt:

```
? @Xbound(6.3412,2.55,6.3411        -> liefert Null
DEFFN Xbound(Wert,Minimum,Maximum)=...
              ...Wert=>Minimum AND Wert<=Maximum
```

BOUNDB() Prüfung auf Absolut-Byte (0 bis 255)

```
Var=BOUNDB(Wert)
```



Durch BOUNDB() kann festgestellt werden, ob sich ***'Wert'*** als Absolut-Byte (0 bis 255) darstellen läßt. Falls nicht, wird die Fehlermeldung *'Bereichsfehler'* ausgelöst.

BOUNDC() Prüfung auf Cardinal-Word (0 bis 65535)

```
Var=BOUNDB(Wert)
```



BOUNDC() ermittelt, ob ***'Wert'*** im *'CARDINAL'*-Bereich (0 bis 65535) liegt. Falls nicht, wird die Fehlermeldung *'Bereichsfehler'* ausgelöst.

BOUNDW() Prüfung auf Signed-Word (-32768 bis +32767)

```
Var=BOUNDW(Wert)
```



BOUNDW() ermittelt, ob ***'Wert'*** im *'WORD'*-Bereich (-32768 bis +32767) liegt. Falls nicht, wird die Fehlermeldung *'Bereichsfehler'* ausgelöst.

11.13. ZUFALLSWERT - ERZEUGUNG

RAND() 16Bit-Integer-Zufallszahl

```
Var=RAND(n)
```



Übergibt eine vorzeichenlose 16Bit-Integer-Zufallszahl aus dem Zahlenbereich 0 (incl.) und ***'n'*** (excl.). Größere Werte als 65535 werden durch `'n' Mod 65535` auf den zulässigen Bereich zurückgerechnet (s. Beispiel zu **XLATE$()**).

Beachten Sie auch das Beispiel zu **STR$()**.

RANDOM() 32Bit-Integer-Zufallszahl

```
Var=RANDOM(n)
```



Übergibt eine 32Bit-Integer-Zufallszahl aus dem Integer-Zahlenbereich 0 (incl.) und 'n' (excl.), wobei ***'n'*** auch negativ sein kann (*signed*).

RANDOMIZE { RA } Zufallszahlengenerator-Init

```
RANDOMIZE [Start]
```



Inistialisiert die Zufallszahlengeneratoren für **RND()**, **RAND()** und **RANDOM()** mit einem Zufallswert. RANDOMIZE wird ohne Parameter intern bei jedem Programmstart ausgeführt. RANDOMIZE 0 hat denselben Effekt.

Bei Verwendung des optionalen Parameters ***'Start'*** wird der Generator mit diesem ***'Start'***-Wert initialisiert. Bei Mehrfachstart mit immer demselben ***'Start'***-Wert wird auch immer dieselbe Zufallszahlenreihe generiert.

RND() Dezimalstellen-Zufallszahl

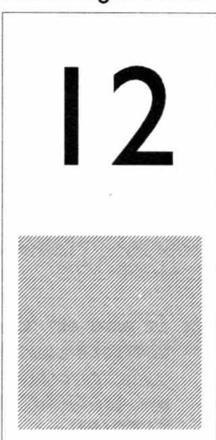
```
Var=RND[(0)]
```



es wird ein 15stelliger Zufalls-Wert im Bereich zwischen 0 (inclusive) und 1 (exclusive) geliefert. Die nachgestellte Klammer ist ein Scheinargument und kann vernachlässigt werden.

Notizen:

Trigonometrie



12. TRIGONOMETRIE

12.1. GRADUMWANDLUNG / PI

DEG() Umwandlung von Bogenmaß in Grad

```
Grad=DEG(Bogenmaß)
```

Rechnet das angegebene ***'Bogenmaß'*** in das Gradmaß (*DEGree*) um (entspricht: ***'Bogenmaß'*** *180/PI).

PI Kreiszahl

```
PI
```

Steht stellvertretend für die konstante Kreiszahl PI, die uns von dem griechischen Philosophen und Mathematiker *Archimedes von Syrakus* (287-212 v.Chr.) beschert wurde. Sie ergibt sich aus der Division des Umfangs eines Kreises durch seinen Durchmesser.

RAD() Umwandlung von Grad in Bogenmaß

```
Radian=RAD(Gradwinkel)
```

Rechnet den angegebenen ***'Gradwinkel'*** in das Bogenmaß (*Radian*) um. (entspricht: ***'Gradwinkel'*** *PI/180).

12.2. PARALLELE TRIGONOMETRIE

ACOS() Arcus-Cosinus

```
Radian=ACOS(Cosinus)
Grad=DEG(ACOS(Cosinus))
```

Es wird ein ***'Cosinus'***-Wert (Ankathete durch Hypothenuse) übergeben, aus dem das dazugehörige Bogenmaß (*Radianwinkel*) zurückgerechnet wird. Wird das Ergebnis in Grad benötigt, muß es mit **DEG(ACOS())** gewandelt werden (entspricht: ACOS() *180/PI).

ASIN() — Arcus-Sinus

```
Radian=ASIN(Sinus)
Grad=DEG(ASIN(Sinus))
```

Es wird ein ***'Sinus'***-Wert (Gegenkathete durch Hypothenuse) übergeben, aus dem das dazugehörige Bogenmaß (*Radianwinkel*) zurückgerechnet wird. Wird das Ergebnis in Grad benötigt, muß es mit **DEG(ASIN())** gewandelt werden (entspricht: ASIN() *180/PI).

ATAN() — Arcus-Tangens

```
Radian=ATAN(Tangens)
Grad=DEG(ATAN(Tangens))
```

Es wird ein ***'Tangens'***-Wert (Gegenkathete durch Ankathete) übergeben, aus dem das dazugehörige Bogenmaß (*Radianwinkel*) zurückgerechnet wird. Wird das Ergebnis in Grad benötigt, muß es mit **DEG(ATAN())** gewandelt werden (entspricht: ATAN() *180/PI).

ATN() — Arcus-Tangens

```
Radian=DEG(ATN(Tangens))
Grad=DEG(ATN(Tangens))
```



ATN() ist identisch mit **ATAN()**. Erläuterungen finden Sie dort.

COS() — genaue Cosinus-Funktion

```
Var=COS(Bogenmaß)
Var=COS(RAD(Gradwinkel))
```



Berechnet den *Cosinus* (Ankathete durch Hypothenuse) für das angegebene ***'Bogenmaß'***. Soll der Winkel als ***'Gradwinkel'*** angegeben werden, muß er durch **RAD(*'Gradwinkel'*)** gewandelt werden (entspricht: ***'Gradwinkel'*** *PI/180).

COSQ() schnelle Cosinus-Funktion

```
Var=COSQ(Gradwinkel)
Var=COSQ(DEG(Bogenmaß))
```

Ermittelt den auf 1/16tel Grad interpolierten *Cosinus* des angegeben ***'Gradwinkels'*** (ca. 10 mal schneller als **COS()**).

Soll der Winkel als ***'Bogenmaß'*** angegeben werden, muß er durch **DEG(*'Bogenmaß'*)** gewandelt werden (entspricht: ***'Bogenmaß'*** *180/PI). Die zweite Variante ist ca. 5 mal schneller als **COS().**

SIN() genaue Sinus-Funktion

```
Var=SIN(Bogenmaß)
Var=SIN(RAD(Gradwinkel))
```

Berechnet den *Sinus* (Gegenkathete durch Hypothenuse) für das angegebene ***'Bogenmaß'***. Soll der Winkel als ***'Gradwinkel'*** angegeben werden, muß er durch **RAD(*'Gradwinkel'*)** gewandelt werden (entspricht: ***'Gradwinkel'*** *PI/180).

SINQ() schnelle Sinus-Funktion

```
Var=SINQ(Gradwinkel)
Var=SINQ(DEG(Bogenmaß))
```

Ermittelt den auf 1/16tel Grad interpolierten *Sinus* des angegeben ***'Gradwinkels'*** (ca. 10 mal schneller als **SIN()**).

Soll der Winkel als ***'Bogenmaß'*** angegeben werden, muß er durch **DEG(*'Bogenmaß'*)** gewandelt werden (entspricht: ***'Bogenmaß'*** *180/PI). Die zweite Variante ist ca. 5 mal schneller als **SIN()**

TAN() Tangens

```
Var=TAN(Bogenmaß)
Var=TAN(RAD(Gradwinkel))
```

Berechnet den *Tangens* (Gegenkathete durch Ankathete) für das angegebene ***'Bogenmaß'***. Soll der Winkel als ***'Gradwinkel'*** angegeben werden, muß er durch **RAD(*'Gradwinkel'*)** gewandelt werden (entspricht: ***'Gradwinkel'*** *PI/180).

12.3. HYPERBOLISCHE TRIGONOMETRIE

ARCOSH() Hyperbel-Area-Cosinus

```
Radian=ARCOSH(Cosinus)
Grad=DEG(ARCOSH(Cosinus))
```

Es wird ein *Hyperbel-'**Cosinus**'* übergeben, aus dem das dazugehörige Bogenmaß (*Radianwinkel*) zurückgerechnet wird. Wird das Ergebnis in Grad benötigt, muß es mit **DEG(ARCOSH())** gewandelt werden (entspricht: ARCOSH() *180/PI).

Der *Hyperbel-Area-Cosinus* ist definiert durch:

ARCOSH(s) = LOG(s+SQR(s^2-1))

ARSINH() Hyperbel-Area-Sinus

```
Radian=ARSINH(Sinus)
Grad=DEG(ARSINH(Sinus))
```

Es wird ein *Hyperbel-'**Sinus**'* übergeben, aus dem das dazugehörige Bogenmaß (Radianwinkel) zurückgerechnet wird. Wird das Ergebnis in Grad benötigt, muß es mit DEG(ARSINH()) gewandelt werden (entspricht: ARSINH()*180/PI).

Der Hyperbel-Area-Sinus ist definiert durch:

ARSINH(s) = LOG(s+SQR(s^2+1))

ARTANH() Hyperbel-Area-Tangens

```
Radian=ARTANH(Tangens)
Grad=DEG(ARTANH(Tangens))
```

Es wird ein *Hyperbel-'**Tangens**'* übergeben, aus dem das dazugehörige Bogenmaß (*Radianwinkel*) zurückgerechnet wird. Wird das Ergebnis in Grad benötigt, muß es mit **DEG(ARTANH())** gewandelt werden (entspricht: ARTANH() *180/PI).

Der *Hyperbel-Area-Tangens* ist definiert durch:

ARTANH(s) = LOG((1+s)/(1-s))/2

COSH() Hyperbel-Cosinus

```
Var=COSH(Bogenmaß)
Var=COSH(RAD(Gradwinkel))
```

Berechnet den *Hyperbel-Cosinus* für das angegebene ***'Bogenmaß'***. Soll der Winkel als ***'Gradwinkel'*** angegeben werden, muß er durch **RAD(*'Gradwinkel'*)** gewandelt werden (entspricht: ***'Gradwinkel'*** *PI/180).

Der *Hyperbel-Cosinus* ist definiert durch:

COSH(w) = EXP(+w)/2 + EXP(-w)/2

SINH() Hyperbel-Sinus

```
Var=SINH(Bogenmaß)
Var=SINH(RAD(Gradwinkel))
```

Berechnet den *Hyperbel-Sinus* für das angegebene ***'Bogenmaß'***. Soll der Winkel als ***'Gradwinkel'*** angegeben werden, muß er durch **RAD(*'Gradwinkel'*)** gewandelt werden (entspricht: ***'Gradwinkel'*** *PI/180).

Der *Hyperbel-Sinus* ist definiert durch:

SINH(w) = EXP(+w)/2 - EXP(-w)/2

TANH() Hyperbel-Tangens

```
Var=TANH(Bogenmaß)
Var=TANH(RAD(Gradwinkel))
```

Berechnet den *Hyperbel-Tangens* für das angegebene ***'Bogenmaß'***. Soll der Winkel als ***'Gradwinkel'*** angegeben werden, muß er durch **RAD(*'Gradwinkel'*)** gewandelt werden (entspricht: ***'Gradwinkel'*** *PI/180).

Der *Hyperbel-Tangens* ist definiert durch:

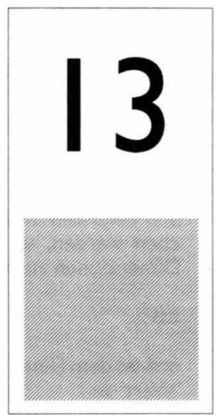
TANH(w) = SINH(w) / COSH(w)

bzw. der *Hyperbel-Cotangens* durch:

coth(w) = COSH(w) / SINH(w)

Kapitel 13

Matrizen-Mathematik



13. MATRIZEN - MATHEMATIK

Die *'Matrizenrechnung'* ist ein Spezialfach der traditionellen Mathematik. Für die analytische Mathematik, die Statistik und z.B. auch bei CAD/CAM- und Raytracing-Berechnungen, der Fakturierung oder aber bei der Lösung linearer Gleichungs- und Differentialgleichungs-Systeme (Statik, Technik) ist sie von großer Bedeutung.

Unter einer Matrix (lat.: *Stammutter*) versteht man allgemein ein zweidimensionales Feld. Stellen Sie sich ein Gitternetz mit der Ausdehnung *'Höhe'* mal *'Breite'* vor. Anstatt der sich ergebenden Flächen zwischen den Linien des Netzes setzen Sie nun numerische Daten ein und erhalten so eine numerische Matrix, also ein rechteckiges Zahlenschema mit der beliebigen Ausdehnung *'Zeilen'* mal *'Spalten'*. Im Gegensatz zur Bildschirmgrafik wird hier die vertikale Dimension *'Zeilen'* (in der Grafik die Y-Richtung) zuerst genannt und ihr folgt die horizontale Dimension *'Spalten'* (in der Grafik die X-Richtung).

'n' Spalten

'm' Zeilen

a(1,1)	a(1,2)	a(1,3)	....	a(1,n)
a(2,1)	a(2,2)	a(2,3)	....	a(2,n)
a(3,1)	a(3,2)	a(3,3)	....	a(3,n)
....	....	....	....	....
a(m,1)	a(m,2)	a(m,3)	....	a(m,n)

Um ein solches Schema übersichtlich und rationell zu speichern und zu bearbeiten wird ein Feld mit Fließkommavariablen eingerichtet, auf welches sich dann die MAT-Befehle sehr vorteilhaft anwenden lassen.

```
DIM Matrix(Zeilen,Spalten)
```

Soll innerhalb dieser Matrix ein bestimmtes Element angesprochen werden, so ist es mit seiner Indizierung in den beiden Dimensionen ausreichend zu identifizieren.

z.B.: `a(5,7)='Wert'`

ordnet dem Element in der 5. Zeile und 7. Spalte den angegebenen 'Wert' zu.

13.1. MATRIZEN - ORGANISATION

MAT ABS { M ABS } Inhalt der Matrix absolut setzen

```
MAT ABS Feld()
```

Löscht alle negativen Vorzeichen der Matrix und erzeugt damit also ausschließlich positive (absolute) Werte.

MAT BASE { M BASE } Start-Index für MAT-Befehle setzen

```
MAT BASE 0
MAT BASE 1
```

MAT BASE bestimmt den Startindex der Felder für weitere MAT-Befehle. Ist allerdings **OPTION BASE 1** eingeschaltet, bleibt MAT BASE 0 wirkungslos, da ja dann keine Null-Elemente in den Feldern existieren. Ist dagegen **OPTION BASE 0** aktiv (Default bei Programmstart), kann mit MAT BASE 0 bestimmt werden, daß auch die Null-Elemente in die Berechnungen mit einbezogen werden. Default-Einstellung bei Programmstart ist MAT BASE 1.

MAT CLR { M C } Inhalt einer Matrix löschen

```
MAT CLR Feld()
```



Setzt alle Elemente der angegebenen Matrix ***'Feld()'*** auf den Wert Null (entspricht: **ARRAYFILL Feld(),0**).

MAT NEG { M NEG } Matrizen-Inhalt negieren

```
MAT NEG Feld()
```



Vertauscht die Vorzeichen des Matrizen-Inhalts. Aus positiven Werten werden negative und umgekehrt (entspricht: **MAT MUL Feld(),-1**).

MAT ONE { M ONE } Einheitsmatrix erzeugen

```
MAT ONE Feld()
```

Unter einer *Einheitsmatrix* versteht man die quadratische Matrix ***'Feld'*** (Zeilen- und Spaltenanzahl sind gleich), in welcher durch MAT ONE die Elemente der fallenden Diagonale (*Feld(1,1)... Feld(n,n)*) mit dem Wert 1 belegt werden. Alle anderen Elemente sind Null. Eine mit einer Einheitsmatrix multiplizierte andere Matrix ergibt immer wieder die andere Matrix, sie verhält sich wie bei einer normalen Multiplikation mit 1.

MAT SET { M S } Matrix mit einem Wert füllen

```
MAT SET Feld()=Wert
```

Durch MAT SET kann die Matrix ***'Feld'*** mit dem angegebenen ***'Wert'*** vorbelegt werden (entspricht: **ARRAYFILL Feld(),Wert**). Es wird unter **OPTION BASE 0** generell auch das Null-Element mit belegt.

MAT TRANS { M T } Matrix transponieren

```
MAT TRANS Ziel()=Quell()
MAT TRANS Feld()
```

Eine Matrix wird transponiert, indem die Zeilen und Spalten vertauscht werden.

Bei der ersten Syntax-Variante wird das zweidimensionale Feld ***'Quell()'*** transponiert und in das entgegengesetzt dimensionierte Feld ***'Ziel()'*** geschrieben. ***'Quell()'*** bleibt dabei unverändert, wogegen der Inhalt von ***'Ziel()'*** verlorengeht.

DIM Feld2(5,3)   DIM Feld1(3,5)

aus:

0	5	10
1	6	11
2	7	12
3	8	13
4	9	14

wird:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

Die zweite Syntax-Variante ist nur auf *quadratische* Matrizen anwendbar. Dabei wird die Matrix ***'Feld()'*** ebenfalls wie beschrieben transponiert, jedoch dann wieder in das Ursprungsfeld ***'Feld()'*** zurückgeschrieben. Der ursprüngliche Inhalt von ***'Feld()'*** geht dann zwar verloren, kann aber nötigenfalls durch eine erneute Transponierung restauriert werden.

MAT TRI { M TRI }   Dreiecksmatrix erzeugen

```
MAT TRI Feld(),Modus
```

Eine Dreicksmatrix wird erzeugt, indem entweder die Elemente oberhalb (***'Modus'***=0) oder die Elemente unterhalb (***'Modus'***=1) der Diagonalen (*Feld(1,1)...Feld(n,n)*) einer quadratischen Matrix ***'Feld()'*** gelöscht werden.

z.B. MAT TRI Feld(),1

aus :

0 , 3 , 6
1 , 4 , 7
2 , 5 , 8

wird :

0 , 3 , 6
0 , 4 , 7
0 , 0 , 8

13.2. MATRIZEN - OPERATIONEN

MAT CPY { M CP } Matrizen (-Ausschnitt) kopieren

```
MAT CPY Ziel()=Quell() [,z3,s3]
MAT CPY Ziel(z1,s1)=Quell() [,z3,s3]
MAT CPY Ziel()=Quell(z2,s2) [,z3,s3]
MAT CPY Ziel(z1,s1)=Quell(z2,s2) [,z3,s3]
```

MAT CPY ermöglicht das Kopieren einer beliebigen ganzen Matrix oder auch einer beliebigen Teil-Matrix in eine zweite beliebige ganze Matrix oder auch beliebige Teil-Matrix. So ist es z.B. möglich, eine größere Anzahl kleinerer Matrizen in einer entsprechend großen *'Mutter'*-Matrix gesammelt anzuordnen und nur bei Bedarf in eine Arbeitsmatrix zu übertragen. Viele **MATxxx**-Befehle (z.B. **MAT INPUT, MAT READ** oder **MAT PRINT**) lassen sich dann sehr rationell auf die Gesamt-Matrix anwenden.

```
MAT CPY Ziel()=Quell()
--------------------------------------
```

Kopiert - bei gleicher Dimensionierung der beiden Felder - die ***'Quell()'***-Matrix komplett in die ***'Ziel()'***-Matrix. Sind die Dimensionierungen der beiden Matrizen verschieden, werden nur soviele Zeile und Spalten kopiert, wie die jeweils kleinere Matrix aufweist.

```
MAT CPY Ziel(z1,s1)=Quell()
----------------------------------------------
```

Kopiert den Inhalt der ***'Quell()'***-Matrix ab deren Startindex in die ***'Ziel()'***-Matrix. Und zwar dort beginnend mit der Zeile ***'z1'*** und der Spalte ***'s1'***. Es werden maximal soviele Zeilen und Spalten kopiert, wie entweder die ***'Quell()'***-Matrix aufweist oder wie maximal in die ***'Ziel()'***-Matrix ab ***'z1,s1'*** hineinpassen.

```
MAT CPY Ziel()=Quell(z2,s2)
-----------------------------------------------
```

Kopiert eine Teilmatrix aus ***'Quell()'*** ab der Zeile ***'z2'*** und der Spalte ***'s2'*** an den Startindex der ***'Ziel()'***-Matrix. Es werden hier nur maximal soviele Zeilen und Spalten kopiert, wie entweder in die ***'Ziel()'***-Matrix hineinpassen oder wie maximal in der ***'Quell()'***-Matrix ab ***'z2,s2'*** zur Verfügung stehen.

```
MAT CPY Ziel(z1,s1)=Quell(z2,s2)
-------------------------------------------------------
```

Das Verfahren dieser Variante ergibt sich logisch aus den beiden vorher beschriebenen Varianten.

```
Option 'z3,s3'
```

Die optionalen Parameter **'z3,s3'** gelten für alle Varianten. Und zwar kann dadurch bestimmt werden, daß maximal **'z3'** Zeilen und **'s3'** Spalten kopiert werden. Stehen soviele Zeilen und Spalten in einer der beiden betroffenen Matrizen nicht zur Verfügung, werden wieder nur soviele Zeilen kopiert, wie maximal möglich oder maximal nötig sind.

MAT INPUT { M INPUT } Matrizen-Inhalt aus Datei lesen

```
MAT INPUT #Kanal,Feld()
```

Durch diesen Befehl kann eine vollständige Matrix aus der Datei mit dem Handle ***'#Kanal'*** vom Festspeicher in das Fließkommafeld ***'Feld()'*** gelesen werden. Die Datei ist dazu mit **'OPEN "I"...'** zu öffnen. Die Daten müssen im ASCII-Format als *'CVS' (CommaSeparatedValue)* vorliegen. Die einzelnen Werte sind durch Kommata oder durch *CR/LF ('Carriage Return' und 'Line Feed' =* **CHR$(13)+CHR$(10)**) zu trennen. MAT INPUT liest ab Dateibeginn zeilenweise soviele Daten, wie möglich oder nötig sind.

```
PRINT #1,''123.234,234.345"
PRINT #1,''345.456,456.567,567.678"
PRINT #1,''678.789,789.890"
```

kann z.B. in eine Matrix mit den Dimensionen

'DIM Feld(2,3)':

123.234 , 234.345 , 345.456
456.567 , 567.678 , 678.789

oder z.B. in eine Matrix mit den Dimensionen

'DIM Feld(3,2)':

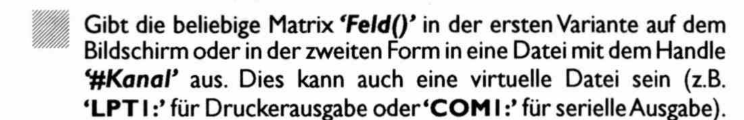
123.234 , 234.345
345.456 , 456.567
567.678 , 678.789

gelesen werden.

In diesen beiden Beispielen würde der Wert '789.890' unberücksichtigt bleiben. Sind dagegen nicht genug Daten vorhanden, wird eine Fehlermeldung ausgelöst.

MAT PRINT { M P } Matrizen-Inhalt ausgeben

```
MAT PRINT Feld() [,Stellen,Realteil]
MAT PRINT #Kanal,Feld() [,Stellen,Realteil]
```



Gibt die beliebige Matrix ***'Feld()'*** in der ersten Variante auf dem Bildschirm oder in der zweiten Form in eine Datei mit dem Handle ***'#Kanal'*** aus. Dies kann auch eine virtuelle Datei sein (z.B. **'LPT1:'** für Druckerausgabe oder **'COM1:'** für serielle Ausgabe).

Die optionalen Parameter ***'Stellen'*** und ***'Realteil'***, die im Gegensatz zu **STR$()** nur beide gemeinsam verwendet werden können, bestimmen durch ***'Stellen'*** die Anzahl der Gesamtstellen (incl. Dezimalpunkt und Vorzeichen), mit denen ein Matrizen-Element dargestellt werden soll und durch ***'Realteil'*** die Anzahl der Stellen, die davon für den Nachkomma-Anteil vorgesehen werden sollen. Dadurch ist ein geordnete Block-Darstellung von Matrizen möglich, auch wenn die einzelnen Werte unterschiedliche Vor- und Nachkommalängen aufweisen.

MAT READ { M READ } Matrizen-Inhalt aus DATAs lesen

```
MAT READ Feld()
```



Dieser Befehl liest aus **DATA**-Zeilen Werte ein und ordnet sie zeilenweise in der Reihenfolge ihres Erscheinens den Elementen der beliebigen Matrix ***'Feld()'*** zu (beachten Sie die Erläuterungen zu **READ/DATA**). Im übrigen gelten hier weitestgehend die Erläuterungen zu **MAT INPUT**.

MAT XCPY { M X } Matrix transponiert kopieren

```
MAT XCPY Ziel([z1,s1])=Quell([z2,s2]) [,z3,s3]
```

MAT XCPY ist prinzipiell identisch mit **MAT CPY** (s. dort), nur daß hier die zu kopierende Matrix gleichzeitig transponiert wird. Das setzt voraus, daß im Normalfall die beiden beteiligten Felder '*umgekehrt*' zu dimensionieren sind (`DIM Ziel(z,s) / DIM Quell(s,z)`).

`'MAT XCPY Ziel()=Quell()'` ist funktionell gleichwertig mit `'MAT TRANS Ziel()=Quell()'`. Im Gegensatz zu **MAT TRANS** müssen hier die Dimensionierungen der beiden Felder nicht passend sein. MAT XCPY liest und kopiert ohne Fehlermeldung immer soviele Elemente wie maximal möglich oder nötig sind. Dafür ist in den '*passenden*' Fällen **MAT TRANS** schneller.

13.3. MATRIZEN-ARITHMETIK

MAT ADD — Matrizen-Inhalte addieren

```
MAT ADD Ziel(),Quell1()+Quell2()
MAT ADD Quellziel(),Wert
MAT ADD Quellziel(),Quell2()
```

Grundsätzlich sind die Matrizen-Additionen vergleichbar mit dem bekannten Additionsbefehl '**ADD a,b**' oder der Zuweisung '**a=b+c**'.

Die erste Variante addiert alle Elemente der Matrix ***'Quell1()'*** mit den entsprechenden Elementen der Matrix ***'Quell2()'*** und legt das Ergebnis wiederum in den entsprechenden Elementen der ***'Ziel()'***-Matrix ab. ***'Quell1()'*** und ***'Quell2()'*** bleiben dabei unverändert. Hierbei wird vorausgesetzt, daß alle drei Matrizen identische Dimensionierungen aufweisen.

Bei der zweiten Variante (vgl. '**ADD a,Wert**' oder '**a=a+Wert**') wird der angegebene beliebige ***'Wert'*** zu allen Elementen der Matrix ***'Quellziel()'*** addiert und das Ergebnis in ***'Quellziel()'*** wieder abgelegt.

Ähnliches geschieht in der dritten Variante, wobei jedoch (identische Dimensionierung in beiden Matrizen vorausgesetzt) jedem Element von ***'Quellziel()'*** das entsprechende Element von ***'Quell2()'*** hinzuaddiert und das Ergebnis in ***'Quellziel()'*** abgelegt wird. ***'Quell2()'*** bleibt bei dieser Aktion unverändert (vgl. '**ADD a,b**' oder '**a=a+b**').

MAT DET { M DET } — Matrizen-Determinante genau berechnen

```
MAT DET Var=Feld() [,Anzahl]
MAT DET Var=Feld(zx,sx),Anzahl
```

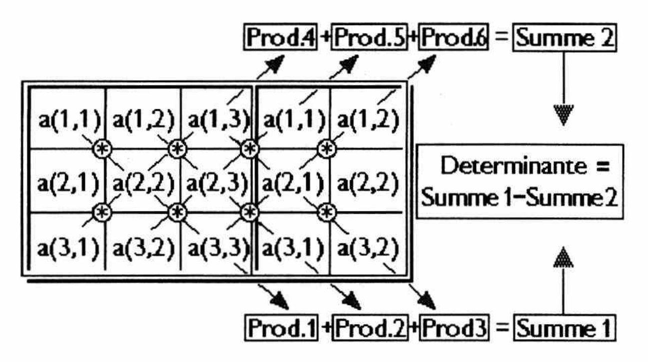
Die Erklärung darüber, was eine *Determinante* ist und worin ihr Sinn und Wert liegt, wäre hier wohl fehl am Platz, da man mit dieser Thematik mehrere Kapitel füllen könnte. Nur soviel: sie ermöglicht eine exakte Identifikation der bezogenen Matrix und ist für die Auflösung von linearen Gleichungssystemen von großer Bedeutung. Ihre einfachste Form ist die zweireihige (2^2) Determinante:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

Die bezogene Matrix muß generell in einem quadratischen Format vorliegen, damit eine Reduktion auf dieses Grundmaß möglich ist.

Bei der Determinanten-Berechnung wird jede Möglichkeit einer Multiplikation von Elementen verschiedener Zeilen und Spalten genutzt, wobei aber gewährleistet sein muß, daß aus jeder Zeile und Spalte immer nur ein Element zur Multiplikation herangezogen wird. Aus der Addition und Subtraktion dieser Produkte nach einem bestimmten Verfahren ergibt sich die Determinante der Matrix.

z.B. 3*3-Matrix:



In der ersten Syntax-Variante wird dieses Verfahren auf die gesamte Matrix ***'Feld()'*** angewandt. Wird der optionale Parameter ***'Anzahl'*** verwendet, so kann damit die Determinante einer Teilmatrix mit ***'Anzahl'*** Zeilen und Spalten bestimmt werden.

Die zweite Variante ermöglicht die Determinanten-Berechnung für eine Teilmatrix von ***'Feld()'***, die mit dem Zeilenindex ***'zx'*** und dem Spaltenindex ***'sx'*** beginnt und die quadratische Ausdehnung von ***'Anzahl'*** Elementen besitzt.

MAT INV — Matrizen-Inverse berechnen

```
MAT INV Ziel(),Quell()
```

Wie mit der Determinante einer Matrix ist auch mit deren Inversion: eine ausführliche Erörterung des Themas sprengt den Rahmen dieses Buches. Wieder nur soviel: multipliziert man die Inverse einer Matrix mit der Matrix selbst, so ergibt sich die Einheitsmatrix (s. **MAT ONE** / **MAT MUL**).

'Ziel()' und ***'Quell()'*** sind zwei quadratische Matrizen mit gleichem Format. Es wird die Inverse der Ursprungsmatrix ***'Quell()'*** berechnet und das Ergebnis dann in ***'Ziel()'*** abgelegt. ***'Quell()'*** bleibt unverändert.

MAT MUL {M M} Matrizen und Vektoren multiplizieren

```
MAT MUL Feld(),Wert
MAT MUL Zielmatrix()=Matrix1()*Matrix2()
MAT MUL Zielmatrix()=S_vektor()*Z_vektor()
MAT MUL Scal=Z_vektor()*S_vektor()
MAT MUL Scal=Z_vektor()*Matrix()*S_vektor()
```

MAT MUL ist ein sehr vielseitiger Befehl, der mehrere Multiplikationsverfahren für Matrizen und Vektoren in sich vereinigt.

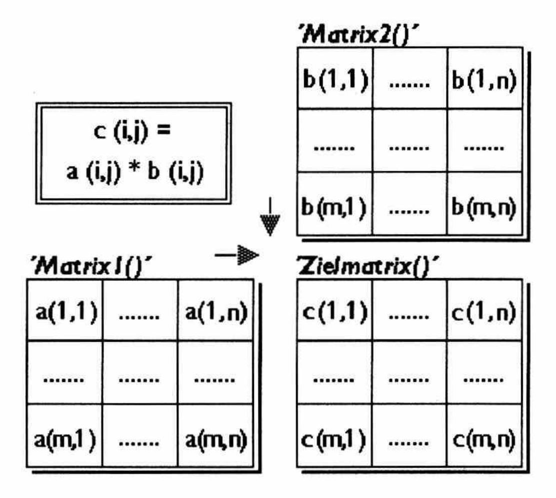
```
MAT MUL Feld(),Wert
-----------------------------------
```

Die erste und einfachste Syntax-Variante multipliziert den Inhalt der angegebenen Matrix ***'Feld()'*** elementeweise mit dem angegebenen ***'Wert'*** und legt das Ergebnis wiederum in den entsprechenden Elementen von ***'Feld()'*** ab.

```
MAT MUL Zielmatrix()=Matrix1()*Matrix2()
----------------------------------------------------------------------
```

Die zweite Variante führt eine Multiplikation der Matrizen ***'Matrix1()'*** und ***'Matrix2()'*** durch und legt die Ergebnisse in den entsprechenden Elementen der ***'Zielmatrix()'*** ab. Dabei ist es erforderlich, daß die Zeilenanzahl von ***'Matrix1()'*** mit der Spaltenanzahl von ***'Matrix2()'*** und umgekehrt die Spaltenanzahl von ***'Matrix1()'*** mit der Zeilenanzahl von ***'Matrix2()'*** identisch ist. Die ***'Zielmatrix()'*** muß so dimensioniert sein, daß ihre Zeilenanzahl mit der von ***'Matrix1()'*** und ihre Zeilenanzahl mit der von ***'Matrix2()'*** identisch ist.

Die Multiplikation erfolgt nach folgendem Schema:

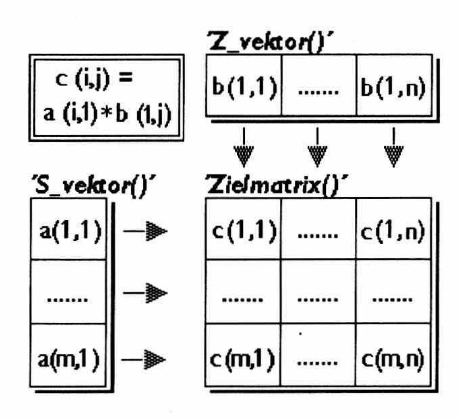


Das Ergebnis-Element ***c1,1*** ergibt sich also aus:

a1,1*b1,1 + a1,2*b2,1 + ...*... +a1,n*bm,1

```
MAT MUL Zielmatrix()=S_vektor()*Z_vektor()
```

Bei dieser Multiplikation wird die ***'Zielmatrix()'*** aus den sog. *'äußeren'* Produkten der Multiplikation des Spaltenvektors ***'S_vektor()'*** mit dem Zeilenvektor ***'Z_vektor()'*** gebildet. Ein Spaltenvektor ist eine eindimensionale Matrix mit ***'n'*** Zeilen, aber nur einer Spalte. Wogegen ein Zeilenvektor eine eindimensionale Matrix mit ***'n'*** Spalten, aber nur einer Zeile ist.
Die Multiplikation erfolgt nach folgendem Schema:



z.B. das Ergebnis-Element c1,1 ergibt sich also aus:

a1,1*b1,1

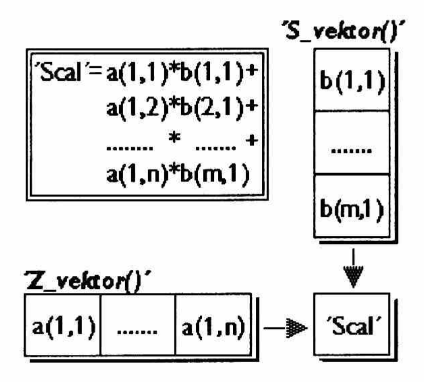
z.B. das Ergebnis-Element cm,2 ergibt sich aus:

am,1*b1,2

```
MAT MUL Scal=Z_vektor()*S_vektor()
```

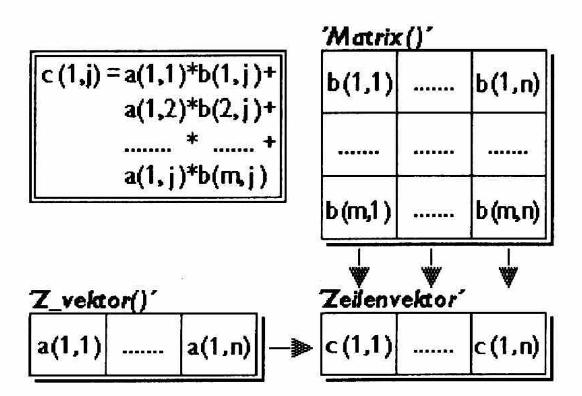
Hier handelt es sich um die Berechnung des sog. *'inneren'* Produkts (*Scalarprodukt*) ***'Scal'*** aus der Multiplikation des Zeilenvektors ***'Z_vektor()'*** und des Spaltenvektors ***'S_vektor()'*** (s.o.).

Die Multiplikation erfolgt nach folgendem Schema:




```
MAT MUL Scal=Z_vektor()*Matrix()*S_vektor()
```

Diese Variante vereinigt zwei verschiedene Multiplikationen in einem Befehl. Zuerst wird die angegebene ***'Matrix()'*** mit dem Zeilenvektor ***'Z_vektor()'*** nach folgendem Schema multipliziert:



z.B. das Ergebnis-Element c1,1 ergibt sich aus:

a1,1*b1,1 + a1,2*b2,1 + ...*... +a1,n*bm,1

z.B. das Ergebnis-Element c1,n ergibt sich aus:

a1,1*b1,n + a1,2*b2,n + ...*... +a1,n*bm,n

Der sich hieraus ergebende neue Zeilenvektor wird dann mit dem Spaltenvektor ***'S_vektor()'*** nach dem in der vorherigen Variante beschriebenen Verfahren multipliziert und ergibt als Endresultat das Scalarprodukt ***'Scal'***.

Aus diesem Verfahren ergibt sich die Notwendigkeit, daß ***'Z_vektor()'*** ebensoviele Zeilen aufzuweisen hat, wie ***'Matrix()'*** an Zeilen und ***'S_vektor()'*** an Zeilenelementen besitzt.

MAT NORM { M NORM } zeilen-/spaltenweise normieren

```
MAT NORM Feld(),0              ->    Zeilennormierung
MAT NORM Feld(),1              ->    Spaltennormierung
```

Auch für MAT NORM gilt: Sinn und Zweck der Matrizen-Normierung ist zu kompliziert, um sie in diesem Rahmen beschreiben zu können. In Stichworten: in der Matrizenrechnung werden normierte Matrizen häufig für die Auflösung komplexer Gleichungssysteme benötigt.

Bei der Normierung wird zuerst die Summe der Quadrate aller Elemente eines Vektors (Zeilen- oder Spaltenvektor) gebildet und dann jedes Element des Vektors mit der Wurzel dieser Summe (dem *'Betrag'* des Vektors) dividiert. Werden die sich daraus ergebenden neuen Werte quadriert, so ergibt die Summe dieser Quadrate des Vektors den Norm-Wert 1.

Die erste Syntax-Variante (`MAT NORM 0`) gliedert dabei die angegebene Matrix ***'Feld()'*** in einzelne Zeilenvektoren und behandelt diese dann nach dem eben beschriebenen Verfahren. Dagegen wird bei der zweiten Variante (`MAT NORM 1`) die Matrix in Spaltenvektoren aufgeteilt und dann die Normierung durchführt. In beiden Fällen wird die Ursprungsmatrix ***'Feld()'*** durch die normierten Werte überschrieben.

MAT QDET { M QDET } Determinante schnell berechnen

```
MAT QDET Var=Feld() [,Anzahl]
MAT QDET Var=Feld(zx,sx),Anzahl
```

Für diesen Befehl gelten die Erläuterungen zu **MAT DET** analog. Es handelt sich prinzipiell um denselben Befehl. MAT QDET ist allerdings geschwindigkeitsoptimiert und somit um ca. 20 Prozent schneller als **MAT DET**. Dagegen hat MAT QDET jedoch den Nachteil der Ungenauigkeit. In den meisten Fällen wird sich dies nicht auswirken, da sich dieses Ungenauigkeiten erst effektiv bemerkbar machen, wenn die Determinante in der Nähe von Null liegt. In diesen Fällen sollte **MAT DET** eingesetzt werden.

MAT RANG { M RANG } Rang einer Matrix ermitteln

```
MAT RANG Var=Feld() [,Anzahl]
MAT RANG Var=Feld(zx,sx),Anzahl
```

Und wieder stehen wir vor dem Dilemma: die Erläuterungen zur Funktion dieses Befehls wären zu umfangreich, um im engen Rahmen dieses Buches abgehandelt zu werden. Wieder in Stichworten: der Rang einer quadratischen Matrix ergibt sich aus der Anzahl der von einander *'unabhängigen'* Zeilen. Eine unabhängige Zeile ist gegeben, wenn sie sich nicht aus der Summe höherstehender Zeilen oder einer Summe von verschiedenen Vielfachen der höherstehenden Zeilen bilden lassen.

z.B.:

```
------------------
 4   6   3   8
------------------
 8  12   6  16  Zeile1*2 ergibt Zeile2
------------------
15   2   5  10  Zeile1+Zeile2 ergibt nicht Zeile3
------------------
27  20  14  34  Zeile2*1.5 + Zeile3 ergibt Zeile4
------------------
 9  11   6  13  Zeile5 kann nicht durch Summierung
------------------  höherer Zeilen gebildet werden
```

Zeile3 und Zeile5 sind also *'unabhängig'* und der Rang dieser Matrix ist somit 2.

In der ersten Syntax-Variante wird die Rang-Bestimmung für die gesamte Matrix ***'Feld()'*** durchgeführt. Wird der optionale Parameter ***'Anzahl'*** verwendet, kann damit der Rang einer Teilmatrix mit ***'Anzahl'*** Zeilen und Spalten bestimmt werden.

Die zweite Variante ermöglicht die Rang-Bestimmung für eine Teilmatrix von ***'Feld()'***, die mit dem Zeilenindex ***'zx'*** und dem Spaltenindex ***'sx'*** beginnt und die quadratische Ausdehnung von ***'Anzahl'*** Elementen besitzt.

MAT RANK { M RANK } Rang einer Matrix ermitteln

```
MAT RANK Var=Feld() [,Anzahl]
MAT RANK Var=Feld(zx,sx),Anzahl
```



MAT RANK ist mit **MAT RANG** identisch. Erläuterungen finden Sie dort.

MAT SUB Matrizen-Inhalte subtrahieren

```
MAT SUB Ziel(),Quell1()-Quell2()
MAT SUB Quellziel(),Wert
MAT SUB Quellziel(),Quell2()
```



Grundsätzlich sind die Matrizen-Subtraktionen vergleichbar mit dem bekannten Subtraktionsbefehl **'SUB a,b'** oder der Zuweisung **'a=b-c'**.

Die erste Variante zieht alle Elemente der Matrix ***'Quell2()'*** von den entsprechenden Elementen der Matrix ***'Quell1()'*** ab und legt das Ergebnis wiederum in den entsprechenden Elementen der ***'Ziel()'***-Matrix ab. ***'Quell1()'*** und ***'Quell2()'*** bleiben dabei unverändert. Hierbei wird vorausgesetzt, daß alle drei Matrizen identische Dimensionierungen aufweisen.

Bei der zweiten Variante (vgl. **'SUB a,Wert'** oder **'a=a-Wert'**) wird der angegebene beliebige ***'Wert'*** von allen Elementen der Matrix ***'Quellziel()'*** subtrahiert und das Ergebnis in ***'Quellziel()'*** wieder abgelegt.

Ähnliches geschieht in der dritten Variante, wobei jedoch (identische Dimensionierung in beiden Matrizen vorausgesetzt) von jedem Element von ***'Quellziel()'*** das entsprechende Element von ***'Quell2()'*** abgezogen und das Ergebnis in ***'Quellziel()'*** abgelegt wird. ***'Quell2()'*** bleibt bei dieser Aktion unverändert (vgl. **'SUB a,b'** oder **'a=a-b'**).

Speicherverwaltung und -Zugriffe



14. SPEICHERVERWALTUNG UND -ZUGRIFFE

14.1. BIT-ARITHMETK

BCHG() — Einzelbit umkehren (an/aus)

```
Var=BCHG(Wert,Bit)
```

Wechselt das angegebene ***'Bit'*** von ***'Wert'*** (`'Wert' XOR (2 ^ 'Bit')`).War das Bit vorher gesetzt ist es anschließend Null und umgekehrt.

BCLR() — Einzelbit löschen

```
Var=BCLR(Wert,Bit)
```

Löscht das angegebene ***'Bit'*** von ***'Wert'*** (`'Wert' AND (NOT 2 ^ 'Bit')`).War das Bit vorher gesetzt, ist es anschließend Null. Null bleibt Null.

BSET() — Einzelbit setzen

```
Var=BSET(Wert,Bit)
```

Setzt das angegebene ***'Bit'*** von ***'Wert'*** (`'Wert' OR (2 ^ 'Bit')`).War das Bit vorher Null, ist es anschließend Eins. Eins bleibt Eins.

BTST() — Einzelbit auf an/aus testen

```
Var=BTST(Wert,Bit)
```

Testet das angegebene ***'Bit'*** von ***'Wert'*** (`-SGN('Wert' AND (2 ^ 'Bit'))`). Ist das Bit gesetzt, wird **TRUE** (-1) geliefert, sonst **FALSE** (0).

AND() / AND Konjunktion zweier Integerwerte

```
Var=AND(Wert1,Wert2)
Var=Wert1 AND Wert2
```

Verknüpft (*'undiert'*) die beiden angegebenen Werte im AND-Modus und liefert das Integer-Ergebnis. Das Ergebnisbit wird nur dann gesetzt, wenn in ***'Wert1'*** *U N D* in ***'Wert2'*** das entsprechende Bit gesetzt ist.

Statt `'Wert1 AND Wert2'` kann auch `'Wert1 & Wert2'` eingesetzt werden.

z.B.:

```
AND(167,236)
```

```
    |->       32Bit binär         <-|  | dezimal
-------------------------------------  |--------
    00000000000000000000000010100111   |     167
AND 00000000000000000000000011101100   | AND 236
-------------------------------------  |--------
=   00000000000000000000000010100100   |   = 164
=====================================  |========
```

EQV() / EQV Äquivalenz zweier Integerwerte

```
Var=EQV(Wert1,Wert2)
Var=Wert1 EQV Wert2
```

Umkehrung zu **XOR()**. Das Ergebnisbit wird nur dann gesetzt, wenn die entsprechenden Bits in ***'Wert1'*** und ***'Wert2'*** entweder beide gesetzt oder beide nicht gesetzt sind.

z.B.:

```
EQV(167,236)
```

```
    |->       32Bit binär         <-|  | dezimal
-------------------------------------  |--------
    00000000000000000000000010100111   |     167
EQV 00000000000000000000000011101100   | EQV 236
-------------------------------------  |--------
=   11111111111111111111111110110100   |   = -76
=====================================  |========
```

IMP() / IMP — Implikation zweier Integerwerte

```
Var=IMP(Wert1,Wert2)
Var=Wert1 IMP Wert2
```



Bei einer *Implikation* ist das Ergebnisbit immer dann Null, wenn das entsprechende Bit in ***'Wert1'*** gesetzt aber in ***'Wert2'*** nicht gesetzt ist. In den anderen drei möglichen Fällen wird das Bit gesetzt.

z.B.:

```
IMP(167,236)
```

```
    |->         32Bit binär          <-|  | dezimal
-------------------------------------     |--------
    00000000000000000000000010100111      |     167
IMP 00000000000000000000000011101100      | IMP 236
-------------------------------------     |--------
=   11111111111111111111111111111100      |   =  -4
====================================      |========
```

NOT — Negation eines Integerwertes

```
NOT Wert
```



Invertiert den angegeben ***'Wert'*** in 32Bit-Breite.

Statt `'NOT Wert'` kann auch `'~ Wert'` eingesetzt werden.

z.B.:

```
NOT 167
```

```
    |->         32Bit binär          <-|  | dezimal
-------------------------------------     |--------
NOT 00000000000000000000000010100111      | NOT 167
-------------------------------------     |--------
=   11111111111111111111111101011000      |  = -168
====================================      |========
```

OR() / OR — incl. Disjunktion zweier Integerwerte

```
Var=OR(Wert1,Wert2)
Var=Wert1 OR Wert2
```



Verknüpft (*'oderiert'*) die beiden angegebenen Werte im OR-Modus und liefert das Ergebnis. Das Ergebnisbit wird schon dann

gesetzt, wenn entweder das entsprechende Bit von ***'Wert1'*** *O D E R* von ***'Wert2'*** gesetzt ist.

Statt `'Wert1 OR Wert2'` **kann auch** `'Wert1 | Wert2'` eingesetzt werden.

z.B.:

```
OR(167,236)
```

```
    |->        32Bit binär        <-|   | dezimal
-------------------------------------   |--------
    00000000000000000000000010100111    |     167
OR  00000000000000000000000011101100    |  OR 236
-------------------------------------   |--------
=   00000000000000000000000011101111    |   = 239
=====================================   |========
```

XOR() / XOR excl. Disjunktion zweier Integerwerte

```
Var=XOR(Wert1,Wert2)
Var=Wert1 XOR Wert2
```



Exclusives Oder. Das Ergebnisbit wird bei XOR nur dann gesetzt, wenn die beiden entsprechenden Bits der angegebenen Werte ungleich, also weder beide gesetzt noch beide nicht gesetzt sind.

z.B.:

```
XOR(167,236)
```

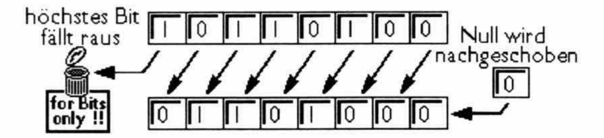
```
    |->        32Bit binär        <-|   | dezimal
-------------------------------------   |--------
    00000000000000000000000010100111    |     167
XOR 00000000000000000000000011101100    | XOR 236
-------------------------------------   |--------
=   00000000000000000000000001001011    |   =  75
=====================================   |========
```

SHL() oder << Bits links verschieben

```
Var=SHL(Wert,Bits)        -> Long-Shift-Left
Var=Wert<<Bits            -> Long-Shift-Left
Var=SHL&(Wert,Bits)       -> Word-Shift-Left
Var=SHL|(Wert,Bits)       -> Byte-Shift-Left
```

Verschiebt den Inhalt von ***'Wert'*** um die Anzahl ***'Bits'*** nach links. Je verschobenem Bit wird ***'Wert'*** dabei mit 2 multipliziert (Integer-

Multiplikation: `Var = Wert * 2 ^ 'Bits')`. Das rechts freiwerdende Bit wird mit 0 gefüllt.

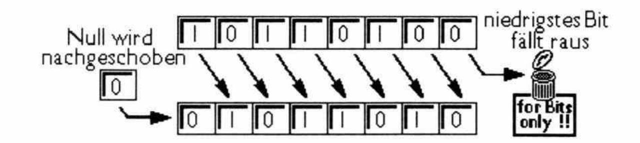


Bei Angabe von ***'&'*** hinter SHL werden nur die ersten 16 Bit (LO-Word) von ***'Wert'*** geshiftet und bei ***'|'*** nur die ersten 8 Bit (LO-Byte). Ist ***'Wert'*** bei SHL|() größer als 8 Bit, so werden als Ergebnis trotzdem nur die untersten 8 Bit der Operation geliefert. Dasselbe gilt für SHL&() (16 Bit), wobei dann jedoch das Bit 15 des Ergebnisses in die Bits 16-31 kopiert wird (*signed*).

SHR() oder >> — Bits rechts verschieben

```
Var=SHR(Wert,Bits)        -> Long-Shift-Right
Var=Wert>>Bits            -> Long-Shift-Right
Var=SHR&(Wert,Bits)       -> Word-Shift-Right
Var=SHR|(Wert,Bits)       -> Byte-Shift-Right
```

Verschiebt den Inhalt von ***'Wert'*** um die Anzahl ***'Bits'*** nach rechts. Je verschobenem Bit wird ***'Wert'*** dabei durch 2 dividiert (Integer-Division: `Var = Wert DIV 2 ^ 'Bits')`. Das links freiwerdende Bit wird mit 0 aufgefüllt. Bei Komplementwerten bleibt also das Vorzeichen gfls. nicht erhalten.

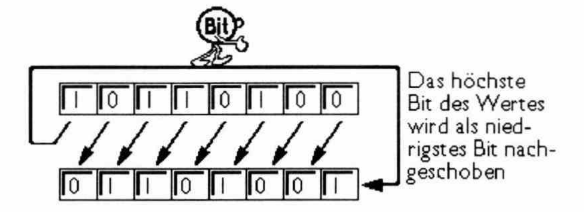


Bei Angabe von ***'&'*** hinter SHR werden nur die ersten 16 Bit (LO-Word) von ***'Wert'*** *geshiftet* und bei ***'|'*** nur die ersten 8 Bit (LO-Byte). Ist ***'Wert'*** bei SHR|() größer als 8 Bit, so werden als Ergebnis trotzdem nur die untersten 8 Bit der Operation geliefert. Dasselbe gilt für SHR&() (16 Bit), wobei dann jedoch das Bit 15 des Ergebnisses in die Bits 16-31 kopiert wird (*signed*).

ROL() — Bits links rotieren

```
Var=ROL(Wert,Bits)          -> Long-Rotate-Left
Var=ROL&(Wert,Bits)         -> Word-Rotate-Left
Var=ROL|(Wert,Bits)         -> Byte-Rotate-Left
```

Rotiert den Inhalt von ***'Wert'*** um die Anzahl ***'Bits'*** nach links. Das jeweils rechts freiwerdende Bit wird dabei mit dem jeweils links herausgeschobenen Bit gefüllt.

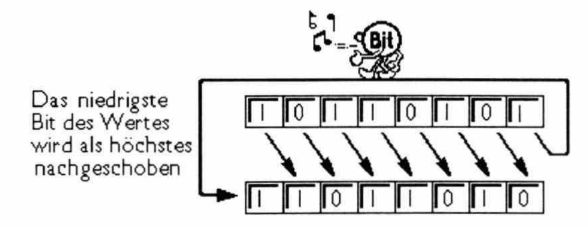


Bei Angabe von ***'&'*** hinter ROL werden nur die ersten 16 Bit (LO-Word) von ***'Wert'*** *rotiert* und bei ***'|'*** nur die ersten 8 Bit (LO-Byte). Ist ***'Wert'*** bei ROL|() größer als 8 Bit, so werden als Ergebnis trotzdem nur die untersten 8 Bit der Operation geliefert. Dasselbe gilt für ROL&() (16 Bit), wobei dann jedoch das Bit 15 des Ergebnisses in die Bits 16-31 kopiert wird (*signed*).

ROR() — Bits rechts rotieren

```
Var=ROR(Wert,Bits)          -> Long-Rotate-Right
Var=ROR&(Wert,Bits)         -> Word-Rotate-Right
Var=ROR|(Wert,Bits)         -> Byte-Rotate-Right
```

Rotiert den Inhalt von ***'Wert'*** um die Anzahl ***'Bits'*** nach rechts. Das jeweils links freiwerdende Bit wird dabei mit dem jeweils rechts herausgeschobenen Bit gefüllt.



Bei Angabe von ***'&'*** hinter ROR werden nur die ersten 16 Bit (LO-Word) von ***'Wert'*** rotiert und bei ***'|'*** nur die ersten 8 Bit (LO-Byte).

Ist ***'Wert'*** bei ROR|() größer als 8 Bit, so werden als Ergebnis trotzdem nur die untersten 8 Bit der Operation geliefert. Dasselbe gilt für ROR&() (16 Bit), wobei dann jedoch das Bit 15 des Ergebnisses in die Bits 16-31 kopiert wird (*signed*).

14.2. BYTE -, WORD - UND LONG - OPERATIONEN

BYTE() LOW-Byte eines Wertes absolut liefern

```
Var=BYTE(Wert)
```

Liefert absolut die untersten 8 Bit von 'Wert'.

CARD() **LOW-Word eines Wertes absolut liefern**

```
Var=CARD(Wert)
```

Liefert absolut (0 bis 65535) die untersten 16 Bit von ***'Wert'***. CARD() ist identisch mit **LOCARD()**, **USHORT()** und **UWORD()**.

HICARD() HI-Word eines Wertes absolut liefern

```
Var=HICARD(Wert)
```

Liefert absolut (0 bis 65535) die obersten 16 Bit von ***'Wert'***.

HIWORD() HI-Word eines Wertes signed liefern

```
Var=HIWORD(Wert)
```

Liefert vorzeichenbehaftet (-32768 bis +32767) die obersten 16 Bit von ***'Wert'***.

LOCARD() LOW-Word eines Wertes absolut liefern

```
Var=LOCARD(Wert)
```



Liefert absolut (0 bis 65535) die untersten 16 Bit von ***'Wert'***. LOCARD() ist identisch mit **CARD()**, **USHORT()** und **UWORD()**.

LOWORD() LOW-Word eines Wertes signed liefern

```
Var=LOWORD(Wert)
```

Liefert vorzeichenbehaftet (-32768 bis + 32767) die untersten 16 Bit von ***'Wert'***.

USHORT() LOW-Word eines Wertes absolut liefern

```
Var=USHORT(Wert)
```



Liefert absolut (0 bis 65535) die untersten 16 Bit von ***'Wert'***. USHORT() ist identisch mit **CARD()**, **LOCARD()** und **UWORD()**.

UWORD() LOW-Word eines Wertes absolut liefern

```
Var=UWORD(Wert)
```

Liefert absolut (0 bis 65535) die untersten 16 Bit von ***'Wert'***. UWORD() ist identisch mit **CARD()**, **LOCARD()** und **USHORT()**.

MAKELONG() Umwandeln zweier Werte in ein Longword

```
Var=MAKELONG(Hiword,Loword)
```



Bildet aus den beiden angegebenen Werten ein Longword. Dazu werden die beiden Werte vorher mit AND 65535 auf 16 Bit formatiert und dann das angegebene *'Hiword'* in die oberen 16 Bit und das *'Loword'* in die unteren 16 Bit des Longwords geschrieben.

SHORT() Wert auf 32Bit erweitern

```
Var=SHORT(Wert)
```

Erweitert ***'Wert'*** vorzeichenbehaftet auf 32 Bit. Das Bit 15 von ***'Wert'*** wird dabei in die obersten 16 Bits des Ergebnisses kopiert. Ist **CARD(Wert)** größer als +32767, ist das Ergebnis von SHORT(Wert) negativ. SHORT() ist identisch mit **WORD()**.

SWAP() HI- und LOW-Word eines Longwords tauschen

```
Var=SWAP(Wert)
```

Vertauscht das LOW-Word (Bits 0-15) von ***'Wert'*** mit dessen HI-Word (Bits 16-31). ***'Wert'*** wird dabei grundsätzlich als 32Bit-Integerwert interpretiert. Evtl. Nachkommastellen werden integriert (s. **INT**). Kleinere ***'Wert'***-Formate als 32 Bit werden auf Long erweitert. Verwechseln Sie diese Funktion bitte nicht mit dem Befehl **SWAP**.

WORD() Wert auf 32Bit erweitern

```
Var=WORD(Wert)
```

Erweitert ***'Wert'*** vorzeichenbehaftet auf 32 Bit. Das Bit 15 von ***'Wert'*** wird dabei in die obersten 16 Bits des Ergebnisses kopiert. Ist **CARD(Wert)** größer als +32767, ist das Ergebnis von WORD(Wert) negativ. WORD() ist identisch mit **SHORT()**.

14.3. SPEICHER - OPERATIONEN

BYTE{} / BYTE{}= 1 Byte absolut lesen / schreiben

```
Var=BYTE{Adresse}
BYTE{Adresse}=Wert
```

Die erste Syntax-Variante liefert das Byte an der angegebenen ***'Adresse'***. Die zweite Variante schreibt den angegebenen Byte-***'Wert'*** an die angegebene ***'Adresse'*** (Segment:Offset).

CARD{} / CARD{}= 2 Byte absolut lesen / schreiben

```
Var=CARD{Adresse}
CARD{Adresse}=Wert
```

Die erste Syntax-Variante liefert das *Absolut*-Word (2 Byte: 0 bis 65535), an der angegebenen ***'Adresse'*** (Segment:Offset). Die zweite Variante schreibt den angegebenen Word-***'Wert'*** an die angegebene ***'Adresse'***.Vorzeichen werden ignoriert. ***'Adresse'*** ist im Format *'Segment:Offset'* anzugeben. CARD{} ist identisch mit **USHORT{}** und **UWORD{}**.

CHAR{} / CHAR{}= 'C'-Text lesen / schreiben

```
Var$=CHAR{Adresse}
CHAR{Adresse}=''Text''
```

Die erste Syntax-Variante liest ab der angegebenen ***'Adresse'*** den Speicher bis zum nächsten auftretenden Nullbyte aus und liefert den bis dahin gelesenen String zurück.

Die zweite Variante schreibt den angegebenen ***'Text'*** (String, Expr$ oder Variableninhalt) ab der angegebenen ***'Adresse'*** in den Speicher und hängt als Abschluß ein Nullbyte an (*'C'-Text / 'null-terminated'*). ***'Adresse'*** ist im Format *'Segment:Offset'* anzugeben.

DOUBLE{} / DOUBLE{}= IEEE-Double lesen / schreiben

```
Realvar=DOUBLE{Adresse}
DOUBLE{Adresse}=Realwert
```

Die erste Syntax-Variante liest ab der angegebenen ***'Adresse'*** acht Byte und liefert sie als IEEE-Fließkommazahl mit doppelter (16stelliger) Genauigkeit zurück. Die zweite Variante schreibt den angegebenen ***'Realwert'*** im 8Byte-IEEE-Double-Format an die angegebene ***'Adresse'*** (Segment:Offset).

DPEEK() 2 Byte signed lesen

```
Var=DPEEK(Adresse)
```

Die Funktion DPEEK() liest das Integer-Word (2 Byte signed: -32768 bis +32767) an der angegebenen ***'Adresse'*** (Segment:Offset).

DPOKE { DP } — 2 Byte signed schreiben

```
DPOKE Adresse,Wert
```

Der Befehl DPOKE schreibt das angegebene Integer-Word ***'Wert'*** vorzeichenbehaftet (-32768 bis +32768) an die angegebene ***'Adresse'*** (Segment:Offset).

INT{} / INT{}= — 2 Byte signed lesen / schreiben

```
Var=INT{Adresse}
INT{Adresse}=Wert
```

Die erste Syntax-Variante liest das Integer-Word (2 Byte signed: -32768 bis +32767) an der angegebenen ***'Adresse'***. Die zweite Variante schreibt den angegebenen Word-***'Wert'*** an die angegebene ***'Adresse'***. ***'Adresse'*** ist im Format *'Segment:Offset'* anzugeben. INT{} ist identisch mit **SHORT{}** und **WORD{}**.

LONG{} / LONG{}= — 4 Byte signed lesen / schreiben

```
Var=LONG{Adresse}
LONG{Adresse}=Wert
```

Die erste Syntax-Variante liest das Integer-Longword (4 Byte signed: -2147483648 bis +2147483647) an der angegebenen ***'Adresse'***. Die zweite Variante schreibt den angegebenen Long-***'Wert'*** an die angegebene ***'Adresse'*** (Segment:Offset). In beiden Fällen reicht auch nur die Angabe der geschweiften Klammer (`Var={Adr}` oder `{Adr}=Wert`). GFA-BASIC interpretiert diese Schreibweise automatisch als Long-Zugriff.

LPEEK() — 4 Byte signed lesen

```
Var=LPEEK(Adresse)
```

Die Funktion LPEEK() liest das Integer-Longword (4 Byte signed: -2147483648 bis +2147483647) an der angegebenen ***'Adresse'*** (Segment:Offset).

LPOKE { LP } — 4 Byte signed schreiben

```
LPOKE Adresse,Wert
```



Der Befehl LPOKE schreibt den angegebenen Integer-Long-*'**Wert**'* vorzeichenbehaftet (-2147483648 bis +2147483647) an die angegebene *'**Adresse**'* (Segment:Offset).

PEEK() — 1 Byte absolut lesen

```
Var=PEEK(Adresse)
```

Die Funktion PEEK() liest das Integer-Byte (absolut: 0 bis 255) an der angegebenen *'**Adresse**'* (Segment:Offset).

POKE { PO } — 1 Byte absolut schreiben

```
POKE Adresse,Wert
```



Der Befehl POKE schreibt das angegebene Integer-Byte *'**Wert**'* vorzeichenlos (absolut: 0 bis 255) an die angegebene *'**Adresse**'* (Segment:Offset).

SHORT{} / SHORT{}= — 2 Byte signed lesen / schreiben

```
Var=SHORT{Adresse}
SHORT{Adresse}=Wert
```

Die erste Syntax-Variante liest das Integer-Word (2 Byte signed: -32768 bis +32767) an der angegebenen *'**Adresse**'*. Die zweite Variante schreibt den angegebenen Word-*'**Wert**'* an die angegebene *'**Adresse**'* (Segment:Offset). SHORT{} ist identisch mit **INT{}** und **WORD{}**.

SINGLE{} / SINGLE{}= — IEEE-Single lesen / schreiben

```
Realvar=SINGLE{Adresse}
SINGLE{Adresse}=Realwert
```

Die erste Syntax-Variante liest ab der angegebenen *'**Adresse**'* vier Byte und liefert sie als IEEE-Fließkommazahl mit einfacher (8stelliger)

Genauigkeit zurück. Die zweite Variante schreibt den angegebenen ***'Realwert'*** im 4Byte-IEEE-Single-Format an die angegebene ***'Adresse'*** (Segment:Offset).

WORD{} / WORD{}= 2 Byte signed lesen / schreiben

```
Var=WORD{Adresse}
WORD{Adresse}=Wert
```

Die erste Syntax-Variante liest das Integer-Word (2 Byte signed: -32768 bis +32767) an der angegebenen ***'Adresse'***. Die zweite Variante schreibt den angegebenen Word-***'Wert'*** an die angegebene ***'Adresse'*** (Segment:Offset). WORD{} ist identisch mit **INT{}** und **SHORT{}**.

USHORT{} / USHORT{}= 2 Byte absolut lesen / schreiben

```
Var=USHORT{Adresse}
USHORT{Adresse}=Wert
```

Die erste Syntax-Variante liefert das *Absolut-Word* (2 Byte: 0 bis 65535), an der angegebenen ***'Adresse'***. Die zweite Variante schreibt den angegebenen Word-***'Wert'*** absolut an die angegebene ***'Adresse'*** (Segment:Offset). Vorzeichen werden ignoriert. USHORT{} ist identisch mit **CARD{}** und **UWORD{}**.

UWORD{} / UWORD{}= 2 Byte absolut lesen / schreiben

```
Var=UWORD{Adresse}
UWORD{Adresse}=Wert
```

Die erste Syntax-Variante liefert das *Absolut-Word* (2 Byte: 0 bis 65535), an der angegebenen ***'Adresse'***. Die zweite Variante schreibt den angegebenen Word-***'Wert'*** absolut an die angegebene ***'Adresse'*** (Segment:Offset). Vorzeichen werden ignoriert. UWORD{} ist identisch mit **CARD{}** und **USHORT{}**.

14.4. BLOCKBEZOGENE OPERATIONEN

BMOVE { BM } Speicherblock kopieren

```
BMOVE Quell,Ziel,Bytes
```

Ab der angegebenen ***'Quell'***-Adresse (Segment:Offset) wird die angegebene Anzahl an ***'Bytes'*** gelesen und an den mit der ***'Ziel'***-Adresse beginnenden Bereich kopiert. Quell- und Zielbereich können sich dabei auch überschneiden.

Hinweis:

*Aufgrund der Segment-Aufteilung eines PC-Speichers ist dieser Befehl auch segmentorientiert. D.h., daß die maximal mit einem Vorgang zu kopierende Blockgröße bei 64 KByte liegt. Dies jedoch auch nur, wenn der Offset zum Segmentstart bei **'Ziel'** und bei **'Quelle'** Null ist. Sonst ist bei **'Bytes'** von den 64 KByte der größere der beiden Offset-Werte abzuziehen. Der Befehl arbeitet bei geraden Adressen schneller als bei ungeraden.*

PEEK$() Speicherblock in Stringvariable kopieren

```
Zielvar$=PEEK$(Quell,Anzahl)
```

Ab der angegebenen ***'Quell'***-Adresse (Segment:Offset) werden ***'Anzahl'*** Bytes gelesen und in die als Empfänger angegebene ***'Zielvar$'*** kopiert. ***'Anzahl'*** ist durch die maximale GFA-Stringlänge von 32767 Zeichen beschränkt. ***'Var$'*** muß nicht vorbereitet werden, da dies von GFA-BASIC in der notwendigen Länge automatisch vorgenommen wird. Beachten Sie bitte auch den Hinweis unter **BMOVE**.

POKE$ Stringvariablen-Inhalt in Speicher kopieren

```
POKE$ Ziel,Quellvar$
```

Der Inhalt der angegebenen ***'Quellvar$'*** wird an die angegebene ***'Ziel'***-Adresse (Segment:Offset) in den Speicher kopiert. Die Länge von ***'Quellvar$'*** sollte nicht größer sein, als die Differenz zwischen Segmentlänge (64 KByte) und Offset. POKE$ überträgt ab ***'Ziel'*** nur soviel Zeichen, wie in das angegebene Segment ab dem Offset noch hineinpassen. Beachten Sie hierzu bitte den Hinweis unter **BMOVE**.

MEMAND { MEMA } Konjunktion zweier Speicherblöcke

```
MEMAND Quell,Ziel,Anzahl
```

Liest ***'Anzahl'*** Bytes ab der angegebenen ***'Quell'***-Adresse (Segment:Offset) und kopiert diese an die ***'Ziel'***-Adresse. Dabei werden die beiden Speicherbereiche im **AND**-Modus miteinander verknüpft. Im Zielbereich sind hinterher nur diejenigen Bits gesetzt, die vorher in beiden Ursprungsbereichen gesetzt waren. Beachten Sie bitte auch den Hinweis unter **BMOVE**.

MEMOR { MEMO } incl. Disjunktion zweier Speicherblöcke

```
MEMOR Quell,Ziel,Anzahl
```

Liest ***'Anzahl'*** Bytes ab der angegebenen ***'Quell'***-Adresse (Segment:Offset) und kopiert diese an die ***'Ziel'***-Adresse. Dabei werden die beiden Speicherbereiche im **OR**-Modus miteinander verknüpft. Im Zielbereich sind hinterher diejenigen Bits gesetzt, die vorher entweder im Quellbereich oder im Zielbereich gesetzt waren. Beachten Sie bitte auch den Hinweis unter **BMOVE**.

MEMXOR { MEMX } excl. Disjunktion zweier Speicherblöcke

```
MEMXOR Quell,Ziel,Anzahl
```

Liest ***'Anzahl'*** Bytes ab der angegebenen ***'Quell'***-Adresse (Segment:Offset) und kopiert diese an die ***'Ziel'***-Adresse. Dabei werden die beiden Speicherbereiche im **XOR**-Modus miteinander verknüpft. Im Zielbereich sind hinterher nur diejenigen Bits gesetzt, die vorher entweder im Quell- aber nicht im Zielbereich oder im Ziel- aber nicht im Quellbereich gesetzt waren (*'oder...aber nicht' = ausschließendes Oder*). Beachten Sie zur Adressangabe bitte auch den Hinweis unter **BMOVE**.

MEMBFILL { MEM } Speicherbereich mit Bytewert füllen

```
MEMBFILL Ziel,Anzahl,Wert
```

Füllt ab der angegebenen ***'Ziel'***-Adresse (Segment:Offset) ***'Anzahl'*** Bytes mit dem angegebenen Byte-***'Wert'*** (absolut: 0 bis 255).

MEMLFILL { MEML } Speicherbereich mit Longwert füllen

```
MEMLFILL Ziel,Anzahl,Wert
```

Füllt ab der angegebenen ***'Ziel'***-Adresse (Segment:Offset) ***'Anzahl'*** Bytes mit dem angegebenen Long-***'Wert'*** (signed: -2147483648 bis +2147483647). Ist ***'Anzahl'*** nicht durch 4 teilbar (1 Longword = 4 Bytes), bleibt das letzte, angeschnittene Longword unberührt. Damit ist gewährleistet, daß der Füllvorgang nicht über die gewünschte Länge hinausgeht.

MEMWFILL { MEMW } Speicherbereich mit Wordwert füllen

```
MEMWFILL Ziel,Anzahl,Wert
```

Füllt ab der angegebenen ***'Ziel'***-Adresse (Segment:Offset) ***'Anzahl'*** Bytes mit dem angegebenen Word-***'Wert'*** (signed: -32768 bis 32767). Ist ***'Anzahl'*** nicht durch 2 teilbar (1 Word = 2 Bytes), bleibt das letzte, angeschnittene Word unberührt. Damit ist gewährleistet, daß der Füllvorgang nicht über die gewünschte Länge hinausgeht.

14.5. SPEICHER - ORGANISATION

MALLOC() System-Speicher-Reservierung

```
Adressvar=MALLOC(Anzahl)
```

Reserviert einen Speicherbereich in der Größe von ***'Anzahl'*** Bytes für den eigenen Bedarf. Im Falle schon vergebener Blöcke liegt dieser Block dann oberhalb des zuletzt mit MALLOC() reservierten oder mit **MSHRINK()** reduzierten Speicherbereichs. Der Bereich ist anschließend gegen System- und GFA-BASIC-Zugriffe geschützt (*alloziert*). Es können hier auch größere Blöcke als 64 KByte für die eigene Speicherverwaltung eingerichtet werden. Es ist allerdings darauf zu achten, daß ein Speicherzugriff nicht über die Segmentgrenzen hinaus liest oder schreibt.

Als Rückgabewert erhält man in der angegebenen ***'Adressvar'*** bei durchgeführter Reservierung die Startadresse (Segment:Offset) des reservierten Bereichs (bei Fehler 0). Wird in ***'Anzahl'*** eine -1 (**TRUE**) übergeben, erhält man in der Rückgabe-Variablen **'Adressvar'** die Größe des momentan für weitere Allozierungen

noch verfügbaren Speichers. Vor Programmende sollte gfls. grundsätzlich **MFREE()** ausgeführt werden, da nach Programmende die MALLOC-Adresse nicht mehr zur Freigabe verfügbar ist und der Speicher dann reserviert bleibt.

MFREE() MALLOC-Speicher wieder freigeben

```
Var=MFREE(Adresse)
```

Gibt einen durch **MALLOC()** reservierten Speicherblock wieder frei. In ***'Adresse'*** (Segment:Offset) wird die Startadresse des freizugebenden Bereichs angegeben (bei **MALLOC()**-Aufruf merken). Wurde die Funktion ohne Fehler ausgeführt, erhält man in der Rückgabe-***'Var'*** eine Null, sonst einen negativen Wert.

MSHRINK() MALLOC-Speicher einschränken

```
Var=MSHRINK(Adresse,Anzahl)
```

Schränkt einen durch **MALLOC()** reservierten Speicherblock ein. In ***'Adresse'*** (Segment:Offset) wird die Startadresse des zu vermindernden Speicherblocks angegeben (bei **MALLOC()**-Aufruf merken).

'Anz' enthält die gewünschte neue Größe des Blocks. Der freiwerdenden Bereich wird an das System zurückgegeben und kann gfls. durch erneutes **MALLOC()** wieder zugeteilt werden. Wurde die Funktion ohne Fehler ausgeführt, erhält man in der Rückgabe-***'Var'*** eine Null, sonst einen negativen Wert.

_PSP Programmsegment-Präfix-Adresse liefern

```
Adressvar=_PSP
```

_PSP ist eine reservierte Variable und liefert die Startadresse (Segment:Offset) des System-Verwaltungsbereichs für das aktuelle Programm (*ProgrammSegmentPräfix* = 256 Bytes). Im Interpreterbetrieb ist dies das *'PSP'* des GFA-Interpreters. Das *'PSP'* liegt direkt am Anfang des ersten Segments, das für den betreffenden Prozess zugewiesen wurde, also direkt vor dem Programm selbst (`Programmstart = _PSP+256`). _PSP ist mit **BASEPAGE** identisch.

Offsets:

+ 0 =	1 Word = Interrupt $21-Aufruf
+ 2 =	1 Word = Segment-Adresse des letzten, für diesen Prozess zugewiesenen Segments
+ 4 =	1 Byte = reserviert
+ 5 =	5 Bytes = *FAR CALL* zum Interrupt $21
+10 =	1 Long = Kopie des Interrupts $22
+14 =	1 Long = Kopie des Interrupts $23
+18 =	1 Long = Kopie des Interrupts $24
+22 =	1 Word = Segment-Adresse des *ProgrammSegmentPräfix* des Aufrufers
+24 bis = +43	20 Bytes = reserviert
+44 =	1 Word = Segment-Adresse des *Environment*-Bereichs
+46 bis = +91	46 Bytes = reserviert
+92 bis = +107	16 Bytes = *FileControlBlock* (FCB) 1
+108 bis = +123	16 Bytes = *FileControlBlock* (FCB) 2
+124 =	1 Long = reserviert
+128 =	1 Byte = Länge der *Kommandozeile* excl. des CR-Bytes am Ende der Komm.
+129 bis = +128	127 Byte = Puffer für *Kommandozeile*

Die *Kommandozeile* ist für die Übergabe von Dateinamen an aufgerufene Programme interessant. Zu Beginn der Komandozeile abglegte Dateinamen (incl. Pfad) werden von vielen Programmen als Startdateien gewertet.

BASEPAGE Programmsegment-Präfix-Adresse liefern

```
Adressvar=BASEPAGE
```

BASEPAGE ist mit **_PSP** identisch. Beachten Sie bitte die Erläuterungen dort.

FREEFONT { FR } externen Font aus dem Speicher löschen

```
FREEFONT
```

Wurde mit **LOADFONT** ein externer Zeichensatz im RAM installiert, so wird dieser durch den Befehl FREEFONT wieder aus dem Speicher gelöscht. Auch **LOADFONT** löst vor seiner Ausführung automatisch ein FREEFONT aus.

LOADFONT { LOADF } externen Font in den Speicher laden

```
LOADFONT ''Fontdatei''
```

Möchten Sie im Grafikmodus einen selbstdefinierten Zeichensatz installieren, so kann dieses durch LOADFONT erreicht werden. Im Textmodus hat LOADFONT dagegen keine Wirkung. Dem Befehl wird der Name einer ***'Fontdatei'*** als Parameter übergeben. Die Freigabe des belegten Speichers erfolgt durch **FREEFONT** (s. dort).

Im *CGA*- und *HGC*-Modus finden Sie die erste Fonthälfte (ASCII 0 - 127) ab $F000:$FA6E. Einen Zeiger auf die Startadresse der oberen Hälfte (ASCII 128 - 255) finden Sie - sofern mit dem DOS-Befehl *GRAFTABL* eine Zeichentabelle installiert wurde - in den Speicherstellen $7C bis $7F (`Adresse=LONG{7C}`).

Ein *EGA/VGA*-Standard-Zeichensatz ist folgendermaßen organisiert:

*Die Gesamtlänge beträgt 4096 Bytes (256 Zeichen * 16 Scanline-Bytes). Ein Zeichen besteht aus 8 Punkten in der Breite und 16 Punkten in der Höhe. Die ersten 16 Bytes des Fonts entsprechen den 16 Scanline-Bytes für das ASCII-Zeichen 0 (Scanline = 8 Muster-Bits je Zeile). Danach schließen sich der Reihe nach weitere 255 ASCII-Zeichen mit wiederum je 16 Bitmuster-Bytes an.*

Den Zeiger auf die Startadresse des internen *ROM-CGA/VGA-Fonts* (4096 Bytes) finden Sie in der Speicherstelle $10C (`Fontadresse=LONG{$10C}`).

LOADFONT erlaubt es allerdings auch, die Fonthöhe (fast) beliebig zu variieren. Die Punkthöhe des zu ladenden Zeichensatzes wird ermittelt, indem GFA-BASIC die Länge der Fontdatei durch 256 teilt. Enthält die Fontdatei also z.B. nur 3072 Bytes, so wird dieser Font mit einer Höhe von 12 Punkten (3072/256) installiert. Ein Font mit einer Länge von z.B. 6144 Bytes würde also demnach mit einer Zeichenhöhe von 24 Punkten installiert.

Weitere Möglichkeiten zur Text-Modifikation finden Sie unter **WINDSET 14** und unter **DEFTEXT**.

FRE() freien Segmentanteil ermitteln

```
Var=FRE([Dummy])
```

Wird als Funktionsargument ein beliebiger ***'Dummy'***-Wert eingesetzt, wird eine *'Garbage Collection'* (Aufräumung des BASIC-Speichers) durchgeführt und anschließend die Größe des noch freien Speichers im aktuell letzten vom GFA-BASIC angeforderten Segment geliefert. D.h., der in der Rückgabe-***'Var'*** gelieferte Wert kann max. 64000 sein. ***'Dummy'*** ist eine Integerwert ohne Bedeutung, der auch weggelassen werden kann. In diesem Fall wird vor der Speicherplatz-Ermittlung keine *'Garbage Collection'* durchgeführt.

Die gesamte momentan verfügbare Speichergröße kann durch **MALLOC(-1)** ermittelt werden.

STACKSIZE { STA } Größe des GFA-Stacks bestimmen

```
STACKSIZE Bytes
```

Mit diesem Befehl kann durch den Parameter ***'Bytes'*** die aktuelle Größe des GFA-internen *Stapelspeichers* (engl.: *stack*) im Bereich von 8000 bis zu 65535 Bytes frei bestimmt werden. Bei verschiedenen GFA-Befehlen ist es notwendig, Daten für den internen Bedarf zwischenzuspeichern oder Adressen und Parameter *'in die Ablage'* zu legen. Diese Ablage funktioniert im wesentlichen wie ein Stapel Papier, auf den ein Blatt obenauf gelegt und bei Bedarf auch wieder heruntergenommen wird. Der sog. *'Stack'* ist ein Speicherbereich, der extra für diese Aufgabe eingerichtet wird. Dabei wird das sog. *'LIFO'*-Prinzip (*LastInFirstOut*) verwendet, d.h.,

es wird bei einem Lesevorgang derjenige Eintrag als nächstes gelesen, der zuletzt gespeichert wurde. GFA-BASIC verwaltet dazu einen Stapel-'*Zeiger*' (engl.: *stackpointer*), der auf den jeweils letzten Eintrag im Stack '*zeigt*'.

Wird z.B. ein Unterprogramm (z.B. **FUNCTION** oder **PROCEDURE**) aufgerufen, so berechnet GFA-BASIC bei jedem Aufruf aus der Adresse des Aufrufes und der Befehlslänge die sog. Rücksprungadresse. Diese Adresse wird dann auf den Stack gelegt. Werden innerhalb der aufgerufenen Routine auch noch lokale Variablen verwendet, werden diese ebenfalls über den Stack verwaltet. Ist die Routine abgearbeitet, wird der Stack wieder geräumt. In der Reihenfolge der Einträge werden die Daten wieder gelesen, verarbeitet und vom Stack gelöscht bzw. der Stackpointer '*heruntergefahren*'.

Der Stack für das GFA-BASIC ist standardgemäß mit einer Größe von 16384 Bytes eingerichtet. Im Normalfall ist diese Größe völlig ausreichend. In den sog. '*rekursiven*' Aufrufen kann es jedoch bei großen Rekursionstiefen passieren, daß die aktuelle Stackgröße nicht ausreicht, um sämtliche Rücksprungadressen und die lokalen Variablen-Pointer zu speichern. Rekursionen sind Prozeduren, die sich aus sich selbst aufrufen, also eine '*Aufruf-Spirale*' bilden.

In diesen Fällen kann eine beliebige Stackgröße bis zu maximal 65535 Bytes eingerichtet werden. Will man Speicher sparen, kann die Stackgröße auch auf minimal 8000 Bytes vermindert werden. Wird die aktuelle Stackgröße durch den Speicherbedarf der Einträge überschritten, wird die Fehlermeldung "*Stapelüberlauf*" ausgegeben.

Zu beachten ist, daß STACKSIZE nicht innerhalb von **PROCEDURE**'s, oder **FOR..NEXT**-Schleifen verwendet werden darf, da sonst die Fehlermeldung '*Nicht in PROC/FOR*' erscheint. Das bedeutet, daß bei streng prozedural strukturierten Programmen eine Stackänderung nur am Programmanfang vor dem '*main*'-Aufruf möglich ist. Die später maximal benötigte Stackgröße sollte dann also schon bei Programmstart bekannt sein.

14.6. ZEIGEROPERATIONEN

ARRPTR() { * } Variablen-/Descriptor-Adresse liefern

```
Var=ARRPTR(Var)         oder    Var=*Var
Var=ARRPTR(Feld())      oder    Var=*Feld()
```

Bei einer String-***'Var$'*** oder einem beliebigen numerischen ***'Feld()'*** wird durch ARRPTR() der dazugehörige *Descriptor* (`ARRPTR(Feld())` oder `ARRPTR (Var$)`) geliefert. Bei numerischen Variablen die Adresse, an welcher der Variableninhalt zu finden ist. `ARRPTR(Var)` ist in diesem Fall identisch mit `VARPTR(Var)` (s. auch Erläuterungen unter 'VARIABLEN-TYPEN')

SWAP { sw } Variablen/Felder/Pointer tauschen

```
SWAP Var1,Var2
SWAP Feld1(),Feld2()
SWAP *Zeiger,Feld()
SWAP Element(x),Element(y)
```

Tauscht die Inhalte zweier gleichartiger Variablen oder Felder (numerische oder alphanumerisch), bzw. einen Zeiger auf einen *Feld-Descriptor* (s.**ARRPTR()**) mit dem *Descriptor* des angegebenen Feldes (z.B. für Feldübergaben an Prozeduren). Bei Feldern wird auch die Dimensionierung vertauscht. Außerdem ist es möglich, einzelne Feldelemente gleichen Typs zu vertauschen (s. Syntax-Variante 4). Verwechseln Sie diesen Befehl bitte nicht mit der **SWAP()**-Funktion.

z.B.:

```
DIM A%(10)
@Routine(*A%())
PROCEDURE Routine(Zeiger%)
  SWAP *Zeiger%,Lokal%()
  ... ab hier ist das globale Feld 'A%()'
  ... innerhalb der Routine unter dem Namen
  ... 'Lokal%()' ansprechbar. Feld 'A%()'
  ... ist nun leer.
  .  ----------------------------
  .
  ... Programmtext des Unterprogramms
  .
  .  ----------------------------
  SWAP *Zeiger%,Lokal%()
  ... ab hier ist Feld 'Lokal%()' wieder
```

```
  ... Feld 'A%()'. Feld 'Lokal%()' ist nun
  ... wieder leer.
RETURN
```

Beachten Sie dazu auch das Beispiel zu **XLATE$**.

VARPTR() {V:} Variablen-Adresse ermitteln

```
Var=VARPTR(Var)
Var=V:Var
```

Liefert bei numerischen Variablen deren Adresse, bzw. bei Stringvariablen die Adresse des ersten Zeichens des betreffende Strings. ***'Var'*** steht hier für jede beliebige Variable, sowie auch für einzelne Feldelemente. Als Abkürzung kann auch ***'V:Var'*** verwendet werden (z.B. `PRINT V:A$`).

14.7. EXPANDED MEMORY - OPERATIONEN (EMS)

EAVAIL { EA } Anzahl freier EMS-Seiten ermitteln

```
EAVAIL Var
```

Die EMS-Befehle **EPUSH**, **EMEMPUSH** und **EMSPUT** lagern Speicherblöcke in das EMS aus. Um in diesen Fällen feststellen zu können, ob im EMS genügend Speicher für die betreffende Operation vorhanden ist, kann mit dem Befehl EAVAIL die momentan verfügbare EMS-Speichergröße ermittelt werden.

Dazu liefert der Befehl in der Rückgabe-***'Var'*** die Anzahl der noch freien EMS-Speicherseiten. Eine EMS-Speicherseite hat eine Größe von 16384 Bytes (16 KByte). Die Größe des freien EMS-Speichers ergibt sich also aus ***'Var'****16384 und dem noch nicht verwendeten Rest der evtl. zuletzt belegten Seite. Diese Restgröße der letzten Seite läßt sich allerdings nur dann exakt feststellen, wenn man sich die Größe der schon ausgelagerten Speicherblöcke programmintern merkt.

EDIR — Ersatz-EMS in einer Datei einrichten

```
EDIR ''[Pfadname]''
```

Im Falle, daß kein EMS vorhanden oder kein weiterer EMS-Speicher verfügbar ist, kann durch EDIR ein simulierter EMS-Speicher innerhalb einer Festspeicher-Datei eingerichtet werden. Diese Datei wird dann vom GFA-BASIC behandelt, als wenn es ein echter EMS-Speicher wäre (nur erheblich langsamer). Ist auf dem Festspeicher (Harddisk, Diskette etc.) genügend Platz vorhanden, ist es so möglich, beliebig große Speicherblöcke dorthin zu verlegen. Dazu wird dem Befehl der gewünschte ***'Pfadname'*** übergeben (z.B. `'C:\EMSDAT\'`). 'Pfadname' kann allerdings auch als Textvariable angegeben werden (z.B. `EDIR path$`). Ein Dateiname darf nicht angegeben werden, bzw. wird ignoriert, da GFA-BASIC im angegebenen Pfad selbstständig eine EMS-Datei anlegt, die vor Programmende auch wieder von GFA-BASIC gelöscht wird.

Sollen die EMS-Befehle wieder auf den echten EMS-Speicher zugreifen, ist `EDIR''''` ohne die Angabe eines Dateinamens zu verwenden. Die EMS-Datei wird dann geschlossen. Sie bleibt jedoch in diesem Fall erhalten und kann nötigenfalls mittels normaler Datei-Befehle (**BLOAD**, **INPUT$()** etc.) ausgelesen werden.

EGET { EG } — freie EMS-Daten-Copy in das DOS-RAM

```
EGET Var:Index[,Feld():Index,Var:''Name'',...
                    ...Feld():''Name'',...]
```

EGET ermöglicht es, auf den durch **EPUSH**, **EMEMPUSH** oder **EMSGET** eingerichteten EMS-Daten-Stack zuzugreifen. Die Variablen- und Feldinhalte können aus dem EMS ausgelesen werden, indem durch EGET entweder der Stack-***'Index'*** oder der durch **EPUSH**, **EMEMPUSH** oder **EMSGET** vergebene ***'Name'*** der betreffenden Daten angegeben wird. ***'Name'*** kann hier auch als Textvariable angegeben werden.

'Index' richtet sich dabei nach der Reihenfolge der Stack-Ablage. Wird als ***'Index'*** eine Null angegeben, wird grundsätzlich der zuletzt auf dem Stack abgelegte Datenblock gelesen. Ein Negativ-***'Index'*** bezieht sich auf das Stack-Ende, während sich ein positiver ***'Index'*** auf den Stack-Anfang bezieht:

Stack-			
Ende =>	**Eintrag 5**	<=	EGET Var:0 oder EGET Var:5
	Eintrag 4	<=	EGET Var:-1 oder EGET Var:4
	Eintrag 3	<=	EGET Var:-2 oder EGET Var:3
Stack-	**Eintrag 2**	<=	EGET Var:-3 oder EGET Var:2
Anfang =>	**Eintrag 1**	<=	EGET Var:-4 oder EGET Var:1

Die Variablen ***'Var'*** und Felder ***'Feld()'***, die mit den gelesenen Daten belegt werden sollen, müssen nicht gesondert vorbereitet werden, da GFA-BASIC zusätzlich zu den Daten auch die Angaben über Variablen- und Feldlängen im EMS abspeichert und die Größen bei der Übergabe entsprechend einrichtet. Es muß allerdings darauf geachtet werden, daß die angegebenen Variablen und Felder dem Typ der zu lesenden Daten entsprechen (Long-Integer zu Long-Integer, Real zu Real etc.)

Die Daten(-blöcke) werden im Gegensatz zu **EPOP** durch EGET im EMS-Stack nicht gelöscht und können also mehrfach gelesen und verwendet werden. Sollen Daten unabhängig von EGET oder **EPOP** gelöscht werden, so kann dazu **EKILL** verwendet werden.

EKILL { EK } — EMS-Daten löschen

```
EKILL [Anzahl] EKILL ''Name''
```

EKILL ermöglicht das Löschen von Daten auf dem EMS-Stack unabhängig von den verschiedenen EMS-Lesebefehlen. Wird EKILL ohne Parameter eingesetzt, so wird der gesamte EMS-Stack gelöscht. Der Parameter ***'Anzahl'*** bewirkt dagegen, daß die letzten (*von 'oben' ausgehend*) ***'Anzahl'*** Stack-Einträge gelöscht werden.

Sollen einzelne Einträge gelöscht werden, so muß ihnen bei der Speicherung durch **EPUSH**, **EMEMPUSH** oder **EMSGET** ein ***'Name'*** zugewiesen worden sein, unter welchem sie dann durch `EKILL ''Name''` angesprochen und selektiv gelöscht werden können. ***'Name'*** kann auch als Textvariable angegeben werden (z.B. `EKILL Nam$`).

EPOP { EPO } — EMS-Daten-Move (lifo) in das DOS-RAM

```
EPOP Var [,Var$,Feld(),Feld$(),...]
```

EPOP liest nach dem *'lifo'*-Prinzip (s. **EPUSH**) Daten bzw. Datenblöcke aus dem EMS-Stack in die angegebenen numerischen

oder String-Variablen ***'Var'*** oder/und in die numerischen oder String-Felder ***'Feld()'***. Die in der Liste angegebenen Variablen und Felder brauchen nicht vorbereitet zu werden, da GFA-BASIC die Angaben über die Variablen- und Feldgrößen zusammen mit den Daten auf den Stack legt und die Vorbereitung, bzw. entsprechende Dimensionierungen intern vor der EPOP-Ausführung automatisch durchführt. Im Gegensatz zu **EGET** löscht EPOP den gelesenen Eintrag vom EMS-Stack.

Achten Sie bitte darauf, daß die angegebenen Variablen und Felder zwar beliebige andere Namen als die bei **EPUSH** verwendeten tragen können, aber in umgekehrter Reihenfolge (!) dem Typ der zu lesenden Daten entsprechen müssen (Long-Integer zu Long-Integer, Real zu Real etc.). Durch die Umkehrung der Reihenfolge wird ermöglicht, daß die bei EPUSH verwendete Variablen- und Feldliste nicht für EPOP gespiegelt werden muß.

z.B. nach

```
EPUSH a%,b$,c&(),d$()
```

ergibt sich folgender Stack:

letzter Eintrag	=>	**d$()**	-Text-Feld
		c&()	- num. Feld
		b$	- Text-Var
		a%	- num. Var
zweiter Eintrag		**???**	- ...
erster Eintrag	=>	**???**	- ...

ein `EPOP a%,b$,c&(),d$()` bewirkt nun, daß die Daten typengerecht wieder zurückgelesen werden. Also die letzte Listenposition ***'d$()'*** wird zuerst mit dem ***'d$()'***-Stackeintrag bedient, dann die vorletzte ***'c&()'***-Position mit dem nächsten ***'c&()'***-Eintrag u.s.w.

EPUSH { EP } DOS-RAM-Daten-Move (lifo) in das EMS

```
EPUSH Var[:''Name''] [,Var$[:''Name''],Feld()...
          ...[:''Name''],Feld$()[:''Name''],...]
```

Dieser Befehl legt Daten und/oder Datenblöcke aus dem DOS-RAM im EMS ab. Das EMS (*Expanded Memory System*) ist durch die EMS-Befehle des GFA-BASICs nur dann ansprechbar, wenn eine Speichererweiterung zur Verfügung steht und bei Systemstart ein entsprechender EMS-Treiber (*EMMxxx.SYS, QUEMM.SYS* etc.) installiert wurde.

Es können dann durch EPUSH beliebige Daten(-blöcke) aus dem Variablenspeicher des GFA-BASICs in das EMS ausgelagert werden, um so für weitere Daten im BASIC-Speicher Platz zu schaffen.

Es empfiehlt sich dabei, nur solche Daten auszulagern, die tatsächlich auch effektiv Speicherplatz verbrauchen, wie z.B. Felder und Stringvariablen. Die mit EPUSH in das EMS transferierten Daten(-blöcke) sind anschließend im DOS-RAM des BASICs nicht mehr vorhanden - es wird bei Variablen ein **CLR** bzw. bei Feldern ein **ERASE** durchgeführt.

'Var$' und ***'Feld()'*** steht hier für die Variablen- und Feldnamen der auszulagernden Variablen und/oder Felder. Durch den optionalen Parameter ***'Name'*** (max. 16 Zeichen), der auf einen Doppelpunkt folgend dem betreffenden ***'Var'***iablen- bzw. ***'Feld()'***-Namen angehängt werden kann, kann den entsprechenden Daten(-blöcken) ein Name zugeordnet werden, unter dem sie im EMS durch **EGET** oder **EKILL** angesprochen werden können. ***'Name'*** kann auch als Textvariable angegeben werden (z.B. `EPUSH Var:Nam1$, Var2:Nam2$,...`).

Das Prinzip eines Stacks erklärt sich dadurch, daß Daten(-blöcke) in der Reihenfolge der Ablage-Operationen (**EPUSH, EMEMPUSH, EMSGET**) auf einen sog. *'Stapel'* (engl.: *stack*) gelegt werden. Der zuletzt abgelegte Variableninhalt bzw. Datenblock liegt also auf diesem Stapel obenauf. Der Begriff *'lifo'* sagt nun aus, daß bei einem Lesezugriff (**EPOP, EMEMPOP, EMSPUT**) zuerst der obenauf liegende Datenblock zur Verfügung steht und danach die weiteren Einträge in der umgekehrten Reihenfolge ihrer Ablage. Das zuletzt auf den Stapel gelegte Daten-*'Paket'* wird nun bei einem Lesevorgang wieder zuerst *'heruntergenommen'* (engl.: *last in, first out = 'lifo'*)

EMEMGET { EMEMG } freie EMS-Block-Copy in das DOS-RAM

```
EMEMGET Ziel,Bytes [:Index]
EMEMGET Ziel,Bytes [:''Name'']
```

Der Befehl EMEMGET hat eine mit **EGET** vergleichbare Aufgabe. Anstatt ausgelagerter Variablen und Felder liest er ganze Speicherbereiche vom EMS-Stack zurück in das DOS-RAM. Wie auch bei **EGET** werden die gelesenen Daten im Gegensatz zu **EMEMPOP** nicht vom Stack gelöscht und sind so also mehrfach verwendbar.

'Ziel' gibt eine Speicheradresse (Segment:Offset) im DOS-RAM an, ab welcher der gelesene Block abgelegt werden soll. Dabei kann durch ***'Bytes'*** zusätzlich bestimmt werden, wieviele Bytes übertragen werden sollen. Wird die Option ***'Name'*** oder ***'Index'*** (Bedeutung s. **EGET** und **EPUSH**) nicht verwendet, so wird der letzte Stackeintrag gelesen. Durch Angabe eines **'Index'** (Trenn-Doppelpunkt beachten) kann unabhängig vom *'lifo'*-Prinzip des Stacks auf beliebige Daten(-blöcke) innerhalb des Stacks zugegriffen werden.

Die ***'Index'***-Zuordnung finden sie unter **EGET** beschrieben. Ebenso verhält es sich mit der Angabe des Parameters ***'Name'***, sofern dem entsprechenden Block bei der Auslagerung durch **EMEMPUSH** ein Name zugewiesen wurde. ***'Name'*** kann auch als Textvariable übergeben und statt des Doppelpunktes vor ***'Index'*** und ***'Name'*** kann auch ein Komma verwendet werden.

Zum allgemeinen Verständnis des EMS-Stacks lesen Sie bitte zusätzlich unter **EGET** und **EPUSH** nach.

EMEMPOP { EMEMPO } EMS-Block-Move in das DOS-RAM

```
EMEMPOP Ziel,Bytes
```

EMEMPOP liest einen Speicherblock nach dem *'lifo'*-Prinzip (s. **EPUSH**) in der angegebenen ***'Bytes'***-Größe vom EMS-Stack und legt ihn im DOS-RAM an der angegebenen ***'Ziel'***-Adresse (Segment:Offset) ab. Der Block wird im Gegensatz zu **EMEMGET** anschließend komplett vom Stack gelöscht. Auch dann, wenn in ***'Bytes'*** eine kleinere Blockgröße als bei dem entsprechenden **EMEMPUSH**-Vorgang angegeben wird.

EMEMPUSH { EM } DOS-RAM-Block-Copy (lifo) in das EMS

```
EMEMPUSH Quell,Bytes [:''Name'']
```

Durch EMEMPUSH können beliebige Speicherbereiche aus dem DOS-RAM auf den EMS-Stack des GFA-BASICs ausgelagert werden. Der Block wird grundsätzlich auf den Stack obenauf gelegt (*'lifo'*-Prinzip s. **EPUSH**).

Durch den Parameter **'Quell'** wird eine Quell-Adresse (Segment:Offset) bestimmt, ab welcher dann soviele Bytes gelesen werden, wie im Parameter **'Bytes'** angegeben wurde. Der gelesene Speicherbereich wird anschließend im DOS-RAM nicht gelöscht, die Daten bleiben erhalten.

Um später durch **EMEMGET** unabhängig vom *'lifo'*-Prinzip auf einen beliebigen Speicherblock innerhalb des EMS-Stacks zugreifen zu können, muß diesem Block vorher durch EMEMPUSH ein Name zugewiesen worden sein. Dies geschieht durch den optionalen Parameter ***'Name'*** (max. 16 Zeichen), dem wahlweise ein Doppelpunkt oder ein Komma als Trennmarkierung vorangestellt wird. ***'Name'*** kann auch als Textvariable angegeben werden (z.B. `EMEMPUSH Quell,Bytes,Nam$`).

Zum allgemeinen Verständnis des EMS-Stacks lesen Sie bitte zusätzlich unter **EGET** und **EPUSH** nach.

EMSGET { EMS } Screen-(Ausschnitt-)Copy in das EMS

```
EMSGET X_links,Y_oben,X_rechts,Y_unten[:''Name'']
```

Bei hochauflösenden Farbgrafiken wird häufig der Fall auftreten, daß durch den Grafik-Speicherbefehl **'GET *x1,y1,x2,y2,Var$*'** die Speicherkapazität einer Stringvariable (32767 Byte) überschritten wird. Dieses Problem kann umgangen werden, indem man entweder den Bildschirm in kleinere Ausschnitte unterteilt, die dann mit mehreren **GET**-Befehlen gesichert werden oder indem man den gesamten Ausschnitt oder Bildschirm ohne Rücksicht auf den Speicherbedarf in einem Schritt durch EMSGET in das EMS auslagert.

Dazu werden dem Befehl in ***'X_links'/'Y_oben'*** die Koordinaten der Ausschnitt-Ecke links-oben und in ***'X_rechts'/'Y_unten'*** die Koordinaten der Ausschnitt-Ecke rechts-unten übergeben. Wird dem Ausschnitt auf den Trenn-Doppelpunkt folgend (kann auch ein Komma sein) ein ***'Name'*** zugewiesen, so kann der Befehl **EMSPUT** später unter Angabe dieses Namens mehrfach auf den Ausschnitt zugreifen, ohne daß der entsprechende Speicherblock im EMS gelöscht wird. ***'Name'*** darf maximal 6 Zeichen lang sein und kann auch als Textvariable angegeben werden.

EMSPUT { EMSP } EMS-(Ausschnitt-)Copy in die Screen

```
EMSPUT X_links,Y_oben[:''Name'']
```

Wurde durch **EMSGET** ein Bildschirm(-Ausschnitt) im EMS abgelegt, so kann dieser durch EMSPUT wieder von dort ausgelesen und in den Bildschirm kopiert werden. Dabei bewirkt die Angabe des optionalen Parameters ***'Name'*** hinter dem Trenn-Doppelpunkt (kann auch ein Komma sein), daß der entsprechende Speicherblock im EMS nicht gelöscht und dann auch mehrfach

nacheinander ausgelesen werden kann. Ein Löschen dieses Blocks ist im EMS dann nur durch den Befehl **EKILL *"Name"*** möglich.

Wird kein ***'Name'*** angegeben, wird der zuletzt mit **EMSGET** im EMS gespeicherte Bildschirm(-Ausschnitt) zurückgelesen und anschließend im EMS gelöscht.

EPARLOAD { EPARL } EMS-Konfiguration aus Datei laden

```
EPARLOAD ''Dateiname''
```

Wurde durch **EPARSAVE** eine komplette EMS-Konfiguration in einer Datei gesichert, kann diese durch EPARLOAD wieder geladen und im EMS installiert werden. Dazu ist dem Befehl der ***'Dateiname'*** (gfls. incl. Pfad) der Konfigurationsdatei zu übergeben. Um diese Daten dann auch nutzen zu können, muß allerdings das ladende Programm entweder Kenntnis über den alten Stack-Aufbau oder die zur Speicherung verwendeten Namen haben. Um die EMS-Organisation nicht zu gefährden, sollte EPARLOAD direkt am Programmanfang eingesetzt werden.

EPARSAVE { EPA } EMS-Konfiguration in Datei speichern

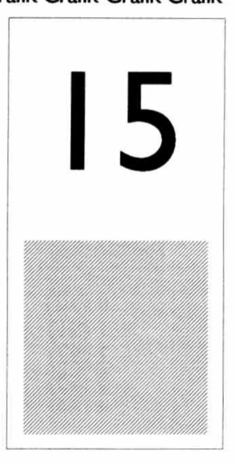
```
EPARSAVE ''Dateiname''
```

Der vom GFA-BASIC belegte EMS-Speicher wird zum Abschluß eines Programms gelöscht. Soll nun vorher - gfls. zwecks Datenübergabe an ein anderes Programm - die momentane EMS-Konfiguration, also der komplette EMS-Stack inclusive aller Organisationsdaten gesichert werden, kann dies durch EPARSAVE erreicht werden.

Dazu wird dem Befehl ein ***'Dateiname'*** (gfls. incl. Pfad) übergeben, unter welchem später die Konfiguration durch **EPARLOAD** wieder zurückgeladen werden kann. Ein anderes Programm kann allerdings nur dann diesen durch **EPARLOAD** zurückgeladenen EMS-Speicher nutzen, wenn es Kenntnis über die alte EMS-Ordnung - also die Stack-Reihenfolge und die gfls. verwendeten Namen - hat. Um die EMS-Organisation im Programmverlauf nicht zu gefährden, sollte EPARSAVE nur am Programmende eingesetzt werden.

Notizen:

Grafik



15. GRAFIK

15.1. GRAFIK - DEFINITIONEN

BOUNDARY { BOU } Rand bei 'P'-Grafikbefehlen an/aus

```
BOUNDARY Flag,Vcol,Hcol
```

Schaltet die Umrandung von *'P'*-Grafikobjekten (**PCIRCLE, POLYFILL, PELLIPSE** etc.) an (***'Flag'*** = 1) oder aus (***'Flag'*** = 0). Zusätzlich kann durch die Parameter ***'Vcol'*** die Vordergrund-Farbe und durch ***'Hcol'*** die Hintergrundfarbe der Umrandung bestimmt werden. Ist durch **DEFLINE** ein durchbrochener Linienstil eingestellte, so werden die Linienfragmente in ***'Vcol'*** und die Zwischenräume in ***'Hcol'*** gezeichnet. Die Umrandung für **PBOX**-Flächen kann bei Bedarf durch den **BOX**-Befehl erzeugt werden.

COLOR { CO } Zeichenfarbe bestimmen

```
COLOR Vcol [,Hcol]
```

Bestimmt durch ***'Vcol'*** (Vordergrundfarbe) und ***'Hcol'*** (Hintergrundfarbe) die Ausführungsfarben für alle linien- und punktezeichnende Grafik-Befehle, für den **TEXT**-Befehl und auch für **PRINT**-Anweisungen im Grafikmodus.

***CGA* 4farbig:**

Farbindex	Farbe
0	Schwarz
1	Türkis
2	Purpur
3	Weiß

***EGA/VGA* 16farbig:**

Farbindex	Farbe
0 ($0)	Schwarz
1 ($1)	Blau
2 ($2)	Grün
3 ($3)	Türkis
4 ($4)	Rot

***VGA* 256farbig:**

Farbindex	Farbe
0 - 15	wieVGA-16farbig
16 - 31	16 Graustufen

Es folgen 9 Farbpaletten
zu je 24 Farben über
Blau - Rot - Gelb

Palette 1 (32 - 55):
100%hell und 100%gesättigt

Palette 2 (56 - 79):
100%hell und 50%gesättigt

Fortsetzung:

***EGA/VGA* 16farbig:**

Farbindex	Farbe
5 ($5)	Purpur
6 ($6)	Braun
7 ($7)	Hellgrau
8 ($8)	Dunkelgrau
9 ($9)	Hellblau
10 ($A)	Hellgrün
11 ($B)	Helltürkis
12 ($C)	Hellrot
13 ($D)	Hellpurpur
14 ($E)	Gelb
15 ($F)	Weiß

***VGA* 256farbig:**

Farbindex	Farbe
Palette 3 (80 -103):	100%hell und 20%gesättigt
Palette 4 (104-127):	50%hell und 100%gesättigt
Palette 5 (128-151):	50%hell und 50%gesättigt
Palette 6 (152-175):	50%hell und 20%gesättigt
Palette 7 (176-199):	20%hell und 100%gesättigt
Palette 8 (200-223):	20%hell und 50%gesättigt
Palette 9 (224-247):	20%hell und 20%gesättigt
248 - 255	Schwarz

Beispiel:

```
SCREEN 18 // oder 14 oder 16 = Farb-VGA, -EGA,
f=1,g=1                          // Startwerte
REPEAT                           // Schleifenstart
  COLOR g                        // Farbe setzen
  FOR i%=-15 TO 15               // 16 Linienmuster
    DEFLINE -2^ABS(i%)           // Muster setzen
    g=g+f                        // Zähler setzen
    IF g>_Y-20 THEN f=-1         // Decrement an
    IF g<0 THEN f=1              // Increment an
    x%=_X/2+SINQ(g)*(_X/2-g)     // X-Koord.1
    x2%=_X/2-SINQ(g)*(_X/2-g)    // X-Koord.2
    y%=_Y/2+COSQ(g)*(_Y/2-g)     // Y-Koord.1
    m2%=_Y/2-COSQ(g)*(_Y/2-g)    // Y-Koord.2
    FOR j%=0 TO 3                // 4 Linien
      LINE x2%+j%,y2%,x%+j%,y%// nebeneinander
      LINE x2%,y2%+j%,x%,y%+j%// zeichnen
    NEXT j%                      // nächste Linie
  NEXT i%                        // nächstes Muster
UNTIL MOUSEK                     //r.Maust.=Abbruch
```

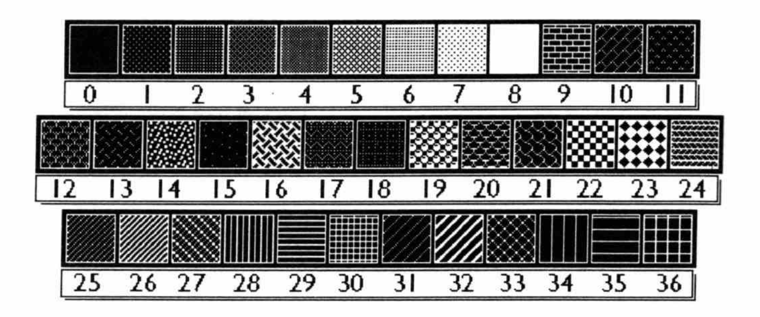
DEFFILL { DEFF } Füllmuster bestimmen

```
DEFFILL Stil
DEFFILL Muster$
```

Legt im Grafikmodus das Füllmuster für deckend gezeichnete Grafik-Objekte (**PBOX, PCIRCLE, PELLIPSE, PRBOX, POLYFILL**) und für Linienbefehle mit einer Liniendicke von mehr als einem Punkt fest. Durch die zweite Syntax-Variante wird außerdem eine eigene Muster-Definition ermöglicht.

'Stil' :

0	kein Muster
1 - 7	Graustufen
8	vollflächig
9 - 24	Punkt-Muster
25 - 36	Strich-Muster



Mit derVariante **DEFFILL Muster$** läßt sich ein eigenes Füllmuster einrichten. Dazu ist in **'Muster$'** ein 8 Byte langer String zu übergeben. Die Bitmuster dieser 8 Bytes beschreiben ein 8x8-Punktraster (1.Byte = 1.Bitmuster-Zeile etc.). Das Muster kann auch als Textausdruck direkt übergeben werden.

z.B.

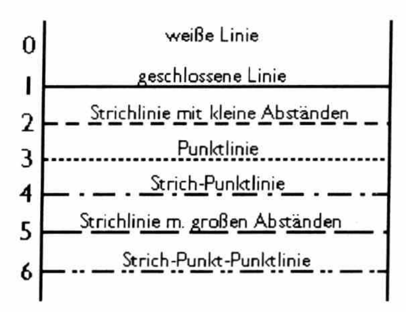
```
DEFFILL "<$ü"+CHR$(231)+CHR$(231)+"ü$<"
```

DEFLINE { DEFL } Linien-Attrribute bestimmen

```
DEFLINE [Stil],[Dicke],[Eckform],[Endform]
```

Durch DEFLINE erfolgt die Festlegung der Attribute, mit denen im Grafikmodus linienzeichnende Befehle ausgeführt werden.

'Stil':



-&X1 bis -&X111111111111111 = selbstdefiniert

Der selbstdefinierte Linien-***'Stil'*** setzt sich aus einem 15 Bit-Wert zusammen, wobei jedes gesetzte Bit einem Punkt in der Linie entspricht.Diese Zahl muß als Minuswert übergeben werden.

'Dicke' legt die eine beliebige Liniendicke fest (in ungeraden 2er-Schritten: 1,3,5...). Ist die Linien-***'Dicke'*** größer als 3, so wird die Linie in dem durch **DEFFILL** eingestellten Füllmuster ausgeführt.

'Eckform' bestimmt die Ausführungsform zweier aufeinandertreffender Linienenden (bei **POLYLINE, DRAW..TO..TO** oder **DRAW$**):

0 = normal
1 = gegehrt
2 = gerundet

'Endform' legt dagegen die Form der Linienenden allgemein fest, also auch bei einzeln gezeichneten Linien (LINE, DRAW..TO ect.)

1 = eckig
2 = quadratisch
3 = gerundet

Die Parameter müssen nicht angegeben werden. Es ist möglich, gezielt nur einzelne Parameter zu ändern. Die Trennkommas zwischen den Parametern sind allerdings immer anzugeben (z.B. `DEFLINE ,5,,2`).

DEFTEXT { DEFT } — Fett-Text an/aus

```
DEFTEXT [?],Art,[?],[?],[?]
```

DEFTEXT ermöglicht im Grafikmodus die Wahl zwischen fetter und normaler Schriftdarstellung.

Außerhalb von GFA-Windows hat dieser Befehl nur Auswirkung auf die mit **TEXT** ausgegebenen Zeichen. Innerhalb von GFA-Windows können auch mit **PRINT** ausgegebene Zeichen beeinflußt werden.

'Art' gibt die gewünschte Darstellung an:

gerader Wert in ***'Art'***	=	normaler Text
ungerader Wert in ***'Art'***	=	fetter Text

Die Besonderheit bei diesem Befehl besteht darin, daß er aus Kompatibilitätsgründen auch im GFA-BASIC unter MSDOS alle fünf im Atari-GFA-BASIC verwendbaren Attribut-Parameter akzeptiert, obwohl nur der zweite Parameter ausgewertet wird. Das Komma vor diesem Parameter muß dazu unbedingt angegeben werden.

Weitere Möglichkeiten zur Text-Modifikation finden Sie unter **WINDSET 14** und **LOADFONT**.

GRAPHMODE { G } — Grafikmodus bestimmen

```
GRAPHMODE Modus
```

Der Parameter ***'Modus'*** entscheidet im Grafikmodus über die Vorgehensweise bei der Verknüpfung von Bildschirmausgaben (**PBOX, LINE, TEXT, PRINT** etc.) mit dem bereits bestehenden Bildschirminhalt.

1 =	*replace*	=	Der alte Inhalt wird durch das neue Objekt komplett ersetzt
2 =	*transparent* (*OR*)	=	Der alte Inhalt wird nur dort überschrieben, wo das neue Objekt Punkte aufweist
3 =	*XOR*	=	Bildpunkte des alten Inhalts werden invertiert, wenn das neue Objekt an gleicher Position Punkte aufweist

4 =	*invers transp.*= *(NOT AND)*	Der alte Inhalt wird dort gelöscht, wo das neue Objekt Punkte aufweist

SETCOLOR { SET } Farbregister einstellen

```
SETCOLOR Reg,Rot,Grün,Blau
```

Es kann die Farbe des mit ***'Reg'*** angegebenen Farbregisters (*CGA* = 0 bis 3, *EGA/VGA* = 0 bis 15) durch Angabe der *RGB*-Farbanteile ***'Rot', 'Grün' und 'Blau'*** (0 bis 63) definiert werden. Die Farbe ist dann durch **COLOR 'Reg'** verfügbar.

SYSCOL { SY } Farbe für Menüs, Fenster etc. bestimmen

```
SYSCOL,Objekt,Vcol,Hcol
```

SYSCOL erlaubt es, den verschiedensten Objekten einer grafischen Benutzeroberfläche - wie z.B. Fenstern, Pulldownmenüs, Alertboxen etc. - unabhängig voneinander beliebige Farben aus der aktuellen Palette zuzuordnen. Voraussetzung dazu ist allerdings, daß vorher durch **SCREEN** ein Bildschirm-Modus eingestellt wurde.

Die Farben werden durch die Parameter ***'Vcol'*** für die Vordergrundfarbe und ***'Hcol'*** für die Hintergrundfarbe des Objektes bestimmt, wobei die Bezeichnungen Vorder- und Hintergrund in einigen Fällen mißverständlich ist. Erläuterungen dazu finden Sie unten. Den Aufbau der Farbpaletten finden Sie unter **COLOR** beschrieben (beachten Sie dazu auch **SETCOLOR**).

Der Parameter ***'Objekt'*** sagt aus, auf welches Element der grafischen Benutzeroberfläche sich die Farbänderung bzw. eine Attribut-Änderung auswirken soll:

0	=	Menüleiste	(vcol , hcol)
1	=	Pulldown-Menüs	(vcol , hcol)
2	=	PopUp-Menüs	(vcol , hcol)
3	=	Fensterrahmen-Fläche	(vcol , hcol)
4	=	Objektumrandung (3D)	(li/ob , re/un)
5	=	Fenster-und PopUp-Text	(wincol , popcol)
6	=	Bildschirm-'Fenster 0'	(vcol , hcol)
7	=	'Fenster 0'-Füllmuster	(muster , 0)
8	=	**ALERT** und **FILESELECT**	(vcol , hcol)

In den Klammern hinter den Objekt-Bezeichnungen ist erkennbar, worauf sich die SYSCOL-Parameter in den einzelnen Fällen beziehen. ***'Vcol'*** und ***'Hcol'*** bedeutet, daß sich die Parameter tatsächlich auf die Vorder- und die Hintergrundfarbe auswirken (bei *'Objekt 6'* bitte **OPENW #0** beachten).

Bei *'Objekt 4'* (3D) ist mit ***'li/ob'*** die Farbe der Objekt-Umrandung an der linken und oberen Seite und mit ***'re/un'*** die Farbe der Objekt-Umrandung an der rechten und unteren Seite gemeint. Bei entsprechender Farbgestaltung (z.B. Hell- und Dunkelgrau oder auch Hell- und Dunkelgrün) kann hiermit ein dreidimensionaler grafischer Effekt erzeugt werden.

Mit *'Objekt 5'* wird durch den ersten Parameter (***'wincol'***) die Text- bzw. Füllmusterfarbe innerhalb der Randobjekte eines aktiven Fensters bestimmt. Die Hintergrundfarbe für die Randobjekte bezieht das aktive Fenster ebenfalls durch ***'Hcol'*** bei *Objekt '3'*. Der zweite Parameter (***'popcol'***) dagegen stellt die Textfarbe für POPUP-Menü-Einträge ein.

'Objekt 7' dagegen hat garnichts mit Farbgestaltung zu tun. Durch den ersten Parameter (***'muster'***) wird hier festgelegt, mit welchem Füllmuster (s. **DEFFILL**) das 'Fenster 0' (s. **OPENW #0**) - also der Bildschirmhintergrund - bei Lage- oder Größenänderungen anderer Fenstern (GFA-Window 1 - 4) in den betroffenen Bereichen restauriert werden soll.

15.2. GRAFIKBEFEHLE

BOX { B } — Linien-Rechteck zeichnen

```
BOX X_links,Y_oben,X_rechts,Y_unten
```

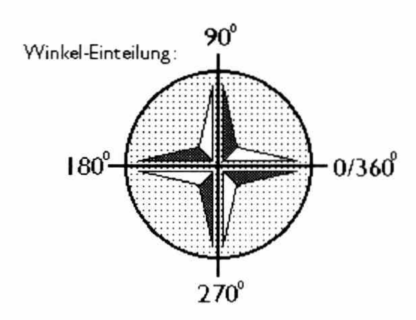
'X_links', 'Y_oben' und ***'X_rechts', 'Y_unten'*** bezeichnen die diagonal gegenüberliegenden Ecken eines Rechtecks, das im Grafikmodus als Linienzug mit den unter **DEFLINE** eingestellten Attributen gezeichnet wird.

CIRCLE { CI } — Linien-Kreis(-Bogen) zeichnen

```
CIRCLE X_mitte,Y_mitte,Radius [,Sw,Ew]
```

Das Koordinatenpaar ***'X_mitte'*** und ***'Y_mitte'*** bestimmt im Grafikmodus den Kreismittelpunkt. ***'Radius'*** beschreibt den

Kreisradius (halber Kreisdurchmesser). Werden die optionalen Parameter ***'Sw'*** und ***'Ew'*** (in Grad: 0 bis 360) verwendet, so wird ein Kreis-Segment mit dem Startwinkel ***'Sw'*** und dem Endwinkel ***'Ew'*** gezeichnet.



CURVE { CU } — 4Punkt-'Bezier'-Kurve zeichnen

```
CURVE Sx,Sy,Mx1,My1,Mx2,My2,Ex,Ey
```

Eine *'Bezier'*-Kurve ist die grafische Darstellung einer mathematischen Formel. Dabei werden die beiden Koordinatenpaare ***'Sx','Sy'*** und ***'Ex','Ey'*** als Start- und Endpunkte einer Linie angesehen. Die beiden Koordinatenpaare ***'Mx1','My1'*** und ***'Mx2','My2'*** stellen dagegen zwei *'Magnet'*-Punkte dar, von denen die Linie zwischen ***'Sx','Sy'*** und ***'Ex','Ey'*** quasi *'angezogen'* wird.

Mit dieser Kurve ist es z.B. relativ leicht möglich, im Grafikmodus auch komplizierte geometrische Figuren mit wenigen CURVE-Befehlen zu zeichnen. Im wesentlichen wird die *'Bezier'*-Kurve bei der platzsparenden und skalierbaren Konstruktion von Schriftzeichen eingesetzt. Statt durch die Definition mehrerer Punktkoordinaten, welche z.B. die Rundungen eines *'S'* beschreiben, kann ein *'S'* in beliebiger Größe dagegen mit nur zwei *'Bezier'*-Kurven konstruiert werden.

Bei der Zeichnung der Kurve wird die aktuelle **DEFLINE**-Einstellung nicht berücksichtigt.

DRAW { DR } — Punkte zeichnen und verbinden

```
DRAW TO Xpos,Ypos
DRAW X1,Y1 [TO X2,Y2 [TO X3,Y3...]]
```

Die erste Syntax-Variante verbindet im Grafikmodus den durch ***'Xpos/Ypos'*** bezeichneten Punkt durch eine Linie mit dem zuletzt

durch **DRAW, PLOT** oder **LINE** gezeichneten Grafik-Punkt. Die **DEFLINE**-Einstellungen werden dabei berücksichtigt.

Die zweite Variante zeichnet entweder einen einzelnen Punkt (vgl. **PLOT**), falls nur ein Koordinatenpaar angegeben wird. Werden mehrere Paare angegeben, so zeichnet der Befehl einen beliebig langen Linienzug durch die in der Koordinaten-Liste angegebene Punkte-Kette.

DRAW "Text" { DR } Plotter(-'Turtle')-Grafik zeichnen

```
DRAW Def$[,Const[,''Def''[,Var[,...]]]]
```

Erlaubt im Grafikmodus eine Plotter-Simulation auf dem Bildschirm (*LOGO-Turtlegrafik*). In ***'Def$'*** bzw. ***"Def"*** können wahlweise als alphanumerischer Ausdruck, als Stringvariable oder Textkonstante *'Turtle'*-Kommandos angegeben werden.

Die Angabe der Entfernungen, Winkel und Koordinaten kann außerdem wahlweise auch als numerischer Ausdruck, als Konstante, als Variable oder innerhalb von ***'Def$'/"Def"*** erfolgen. Dabei ist die Anzahl der durch Kommas getrennten Einzelanweisungen beliebig (max. Eingabezeilenlänge = 255 Zeichen). Das Komma kann in den meisten Fällen auch vernachlässigt werden.

Als Kommandos sind folgende Kürzel gültig: (*'n'* enthält den jeweils anzugebenden Wert, *'x,y'* die betreffenden Koordinaten)

fd n :	bewege 'Stift' um *'n'* Pixel vorwärts
bk n :	bewege 'Stift' um *'n'* Pixel rückwärts
sx n :	Scalierung aller bei *'fd'* und *'bk'* angegeben Werte in X-Richtung mit dem Wert *'n'* (0 = Scalierung aus)
sy n :	Scalierung aller bei *'fd'* und *'bk'* angegeben Werte in Y-Richtung mit dem Wert *'n'* (0 = Scalierung aus)
lt n :	drehe 'Stift' um *'n'* Grad nach links
rt n :	drehe 'Stift' um *'n'* Grad nach rechts
tt n :	setze 'Stift' absolut in Richtung *'n'* Grad (Gradeinteilung s. **SETDRAW**)
ma x,y :	bewege 'Stift' (pu) absolut nach *'x,y'* (s. auch **SETDRAW**)
da x,y :	bewege 'Stift' zeichnend (pd) zu der relativen Position *'x,y'*
mr x,y :	bewege 'Stift' (pu) relativ nach *'x,y'*
dr x,y :	bewege 'Stift' zeichnend (pd) zu der relativen Position *'x,y'*

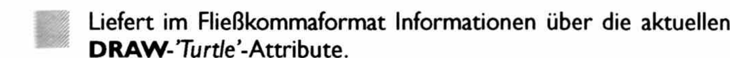
co n : Linienfarbe *'n'* einstellen (s. **COLOR**)
pu : 'Stift' anheben (Stift *'schwebt'*)
pd : 'Stift' aufsetzen

z.B.: (s. auch unter **SETDRAW**)

```
A|=40
A$=''ma320,200
DRAW A$,''tt45 pd fd70 rt90 fd40 pu fd20 pd''
DRAW ''fd'',A|,''lt135 bk'',4*60-140,''rt315 fd 170"
```

DRAW() — Plotter(-'Turtle')-Attribute ermitteln

```
Var=DRAW(Index)
```



Liefert im Fließkommaformat Informationen über die aktuellen **DRAW**-*'Turtle'*-Attribute.

'Index':
0 = X-Position
1 = Y-Position
2 = Winkel in Grad
3 = X-Skalierungsfaktor
4 = Y-Skalierungsfaktor
5 = Pen-Flag (-1=pu; 0=pd)

ELLIPSE { ELL } — Linien-Ellipse(n-Bogen) zeichnen

```
ELLIPSE Xmitte,Ymitte,Xrad,Yrad [,Sw,Ew]
```

'Xmitte' und ***'Ymitte'*** legen im Grafikmodus den Mittelpunkt der zu zeichnenden Ellipse fest. ***'Xrad'*** ist der Ellipsenradius in horizontaler Richtung und ***'Yrad'*** der Radius in vertikaler Richtung. Zur Winkeleinteilung für die Option ***'Sw'*** und ***'Ew'*** beachten sie bitte die Ausführungen zu **CIRCLE**.

FILL { FI } — Flächen mit Muster füllen

```
FILL Xpos,Ypos [,Farbe]
```

'Xpos' und ***'Ypos'*** geben die absolute Lage des Bildschirmpunktes an, bei welchem im Grafimodus der Füllvorgang begonnen werden soll. Dabei wird das aktuell durch **DEFFILL** eingestellte Füllmuster verwendet.

Mit dem optionalen Parameter ***'Farbe'*** kann ein Farbwert angegeben werden, der dann bewirkt, daß ausschließlich Bildschirmpunkte mit der angegebenen Farbe als Füllbegrenzung gewertet werden. Alle anderen Punkte werden gefüllt. Hat z.B. der Bildpunkt ***'Xpos'/'Ypos'*** die angegebene ***'Farbe'***, wird der Füllvorgang sofort abgebrochen.

Bei eingeschaltetem Clipping (s. **CLIP**) wird generell nur bis an die Grenzen des aktuellen **CLIP**-Ausschnitts gefüllt.

Beachten Sie auch das Beispiel zu **STR$()**.

LINE { LI } — Linie zeichnen

```
LINE Xpos1,Ypos1,Xpos2,Xpos2
```

Die Koordinatenpaare ***'Xpos1', 'Ypos1'*** und ***'Xpos2', 'Ypos2'*** werden im Grafikmodus durch eine gerade Linie verbunden. Dabei werden die zuletzt durch **DEFLINE** und **COLOR** ausgewählten Attribute berücksichtigt.

PBOX { PB } — gefülltes Rechteck zeichnen

```
PBOX X_links,Y_oben,X_rechts,Y_unten
```

'X_links', 'Y_oben' und ***'X_rechts', 'Y_unten'*** bezeichnen die diagonal gegenüber liegenden Ecken eines Rechtecks, das im Grafikmodus als gefüllte Fläche mit dem aktuell durch **DEFFILL** eingestellten Füllmuster gezeichnet wird.

PCIRCLE { PC } — gefüllten Kreis(-Ausschnitt) zeichnen

```
PCIRCLE X_mitte,Y_mitte,Radius [,Sw,Ew]
```

'X_mitte' und ***'Y_mitte'*** bestimmen den Mittelpunkt der im Grafikmodus zu zeichnenden Kreisfläche. Der Parameter ***'Radius'*** bestimmt dabei den Radius (halben Kreisdurchmesser). Werden die optionalen Parameter ***'Sw'*** und ***'Ew'*** (in Grad: 0 bis 360) verwendet, so wird ein *'Torten'*-Segment mit dem Startwinkel ***'Sw'*** und dem Endwinkel ***'Ew'*** gezeichnet. Die Winkeleinteilung finden Sie unter **CIRCLE** beschrieben.

PELLIPSE { PE } gefüllten Ellipse(n-Ausschnitt) zeichnen

```
PELLIPSE Xmitte,Ymitte,Xrad,Yrad [,Sw,Ew]
```

'Xmitte' und ***'Ymitte'*** legen den Mittelpunkt der im Grafikmodus zu zeichnenden Ellipsenfläche fest. ***'Xrad'*** ist der Ellipsenradius in horizontaler Richtung und ***'Yrad'*** der Radius in vertikaler Richtung. Zur Winkeleinteilung (in Grad: 0 bis 360) für die Option ***'Sw'*** und ***'Ew'*** beachten sie bitte die Ausführungen zu **ELLIPSE** und **PCIRCLE**.

PLOT { PL } einzelnen Bildschirmpunkt zeichnen

```
PLOT Xpos,Ypos
```

Das Kordinatenpaar ***'Xpos'*** und ***'Ypos'*** bestimmt im Grafikmodus die Lage eines Bildschirmpunktes, der in der durch **COLOR** bestimmten Farbe gezeichnet werden soll. Dabei stehen die angegebenen Koordinaten entweder in Relation zur absoluten linken, oberen Bildschirmecke oder in Fenstern zur linken, oberen Fensterecke oder zu dem durch **CLIP OFFSET** bestimmten Nullpunkt. **PLOT** wird von der Linienbreite-Definition durch **DEFLINE** nicht beeinflußt.

POLYFILL { POLYF } gefülltes Vieleck zeichnen

```
POLYFILL Pkte,Xp(),Yp() [OFFSET Xoff,Yoff]
```

Es wird im Grafikmodus ein beliebig geformtes Polygon gezeichnet ***'Pkte'*** bestimmt dabei die Anzahl der zu verbindenden Punkte (max. 128). Die vorher zu dimensionierenden Integer-Felder ***'Xp()'*** und ***'Yp()'*** enthalten in analoger Reihenfolge jeweils die X- und Y-Koordinaten der Eckpunkte. Dabei enthält das erste Element des Feldes (*'Xpos(0)'* bzw. *'Ypos(0)'* bei **OPTION BASE 0**) jeweils die entsprechende Koordinate des ersten Eckpunktes. Es werden nun aufsteigend soviele Koordinatenpaare aus den Feldern verwertet, wie in ***'Pkte'*** angegeben wurde. Der Start- und der Endpunkt des Polygons werden automatisch miteinander verbunden und anschließend die zwischen den Polygon-Linien eingeschlossenen Flächen mit dem in **DEFFILL** angegebenen Füllmuster ausgefüllt.

Mit der Option ***OFFSET 'Xoff','Yoff'*** kann der Linienzug durch Angabe eines Pixel-Offsets unter Beibehaltung der ursprünglichen Feld-Inhalte in die entsprechende Richtung verschoben werden.

POLYLINE { POL } — Linien-Vieleck zeichnen

```
POLYLINE Pkte,Xp(),Yp() [OFFSET Xoff,Yoff]
```

Es gelten die gleichen Ausführungen wie zu **POLYFILL**, nur daß hier der gezeichnete Linienzug nicht gefüllt wird. **DEFLINE** wird hierbei berücksichtigt.

PRBOX { PRB } — gefülltes Rundeck zeichnen

```
PRBOX X_links,Y_oben,X_rechts,Y_unten
```

Es gelten die gleichen Ausführungen wie zu **PBOX**. Die Rechteckfläche wird jedoch mit runden Ecken gezeichnet.

PSET { PS } — Punkt zeichnen incl. Farbbestimmung

```
PSET Xpos,Ypos,Farbe
```

PSET ist prinzipiell identisch mit **PLOT** (s. dort), jedoch langsamer.Anders als bei **PLOT** muß hier zusätzlich zu den Koordinaten angegeben werden, in welcher ***'Farbe'*** der Punkt gezeichnet werden soll (s. **COLOR**). Dadurch ist es jedoch möglich, einzelne Punkte in verschiedenen Farben zu zeichnen, ohne die allgemeine Farbeinstellung dazu ändern zu müssen.

RBOX { RB } — Linien-Rundeck zeichnen

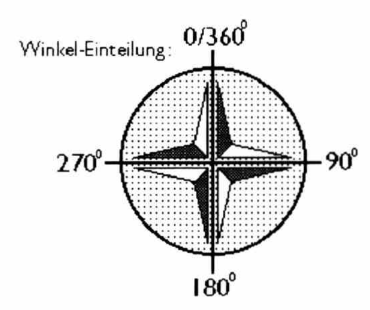
```
RBOX X_links,Y_oben,X_rechts,Y_unten
```

Es gelten die gleichen Ausführungen wie zu **BOX**. Das Rechteck wird jedoch mit runden Ecken gezeichnet.

SETDRAW { SETD } — DRAW-'Turtle' positionieren

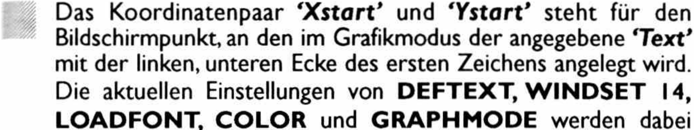
```
SETDRAW Xpos,Ypos,Grad
```

Setzt im Grafikmodus den Stift für Plotter(-*'Turtle'*)-Grafik auf die absolute Position ***'Xpos','Ypos'*** und dreht die sog. *'Turtle'* in die durch ***'Grad'*** angegebene Richtung .



Beispiel: (ohne Erklärung - einfach abtippen)

```
SCREEN 18
GRAPHMODE 3
FOR j%=0 TO 1
 FOR k%=60 TO _Y-40 STEP 40
  FOR i%=60 TO _X-80 STEP 40
   SETDRAW i%,k%,i%+90
   DRAW ''pd rt90 fd10 rt90 fd15 lt45 fd7 lt45 fd20''
   DRAW ''lt45 fd7 lt45 fd5 lt90 fd10 lt90 fd5 rt90''
   DRAW ''fd10 rt90 fd15 rt90 fd30 rt90 fd20 rt45''
   DRAW ''fd14.1 rt45 fd30 rt45 fd14.1 rt45 fd20 pu''
   DRAW ''fd15 pd fd20 rt90 fd10 rt90 fd15 lt45 fd7''
   DRAW ''lt45 fd5 lt90 fd10 rt90 fd10 rt90 fd10 lt90''
   DRAW ''fd20 rt90 fd10 rt90 fd40 rt45 fd14.1 rt45''
   DRAW ''fd20 pu fd15 pd fd15 rt45 fd14.1 rt45 fd40''
   DRAW ''rt90 fd10 rt90 fd20 lt90 fd15 lt90 fd20 rt90''
   DRAW ''fd10 rt90 fd40 rt45 fd14.1 pu rt135 fd15 pd''
   DRAW ''fd5 lt90 fd15 lt90 fd5 lt45 fd7 lt45 fd5lt45''
   DRAW ''fd7 pu''
  NEXT i%
 NEXT k%
NEXT j%
```



TEXT { T } — Text im Grafikmodus ausgeben

```
TEXT Xstart,Ystart,''Text''
```

Das Koordinatenpaar ***'Xstart'*** und ***'Ystart'*** steht für den Bildschirmpunkt, an den im Grafikmodus der angegebene ***'Text'*** mit der linken, unteren Ecke des ersten Zeichens angelegt wird. Die aktuellen Einstellungen von **DEFTEXT, WINDSET 14, LOADFONT, COLOR** und **GRAPHMODE** werden dabei berücksichtigt.

'Text' kann sowohl direkt als Text, als Stringvariable oder als zusammengesetzter Textausdruck (`A$+Str$(''1'')`) angegeben werden.

15.3. GRAFIKBILDSCHIRM-OPERATIONEN

_ADAP aktuellen Grafik-Adapter ermitteln

```
Var=_ADAP
```

Durch diese reservierte Variable kann festgestellt werden, welche Grafik-Karte momentan angeschlossen ist.

0	=	*MDA*	- Mono-Textkarte
1	=	*HGC*	- Mono-Hercules-Karte
2	=	*CGA*	- Color-Grafik-Karte
3	=	*EGA*	- Color-Grafik-Karte
4	=	*VGA*	- Color-Grafik-Karte
5	=	*???*	- evtl. Super-VGA

_C mögliche Farb-Anzahl ermitteln

```
Var=_C
```

Diese reservierte Variable enthält die Information über die zur Zeit möglichen Farben:

MDA	=	0
HGC	=	2
CGA	=	4
EGA/VGA (je nach Modus)	=	2 bis 16
VGA (SCREEN-Modus 19)	=	256

_MD aktuellen SCREEN-Modus ermitteln

```
Var=_MD
```

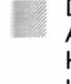
Durch diese reservierte Variable kann ermittelt werden, welcher Bildschirm-Modus momentan aktiviert ist. Der hier gelieferte Wert entspricht dem in **SCREEN** anzugebenden Modus-Wert (s. dort).





_X aktuelle Bildschirm-/Fenster-Breite ermitteln

```
Var=_X
```



Die reservierte Variable _X liefert die aktuell verfügbare horizontale Auflösung des Bildschirms. Dies ist im Grafikmodus die Pixel-Breite (z.B. bei *SCREEN 18:* 640) und im Textmodus die Textraster-Breite (z.B. bei *SCREEN 3:* 80). Ist durch **OPENW#** ein Fenster aktiviert worden, so wird die verfügbare Pixel-Breite der Fenster-Arbeitsfläche geliefert.

Beachten Sie auch die Beispiele zu **STR$()** und **SETDRAW**.

_Y aktuelle Bildschirm-/Fenster-Höhe ermitteln

```
Var=_Y
```

Die reservierte Variable _Y liefert die aktuell verfügbare vertikale Auflösung des Bildschirms. Dies ist im Grafikmodus die Pixel-Höhe (z.B. bei *SCREEN 18:* 480) und im Textmodus die Textraster-Höhe (z.B. bei *SCREEN 3:* 25). Ist durch **OPENW#** ein Fenster aktiviert worden, so wird die verfügbare Pixel-Höhe der Fenster-Arbeitsfläche geliefert.

Beachten Sie auch die Beispiele zu **STR$()** und **SETDRAW**.

CLIP { CLI } Grafikausgabe-Bereich/-Nullpunkt bestimmen

```
CLIP Xli,Yob,Breite,Höhe [OFFSET X,Y]
CLIP Xli,Yob TO Xre,Yun [OFFSET X,Y]
CLIP #Nummer [OFFSET X,Y]
CLIP OFFSET X,Y
```



CLIP ermöglicht im Grafikmodus die Bestimmung eines Bildschirmrechtecks, auf welches dann (außer **PUT**) sämtliche Grafikausgaben begrenzt werden. Alle Teile von Grafikausgaben, die außerhalb der Grenzen dieses Rechtecks liegen, werden nicht gezeichnet.

In der ersten Syntax-Variante gibt das Parameter-Paar ***'Xli'*** und ***'Yob'*** die Position der linken, oberen Ecke, sowie ***'Breite'*** und ***'Höhe'*** die Pixel-Ausdehnung des CLIP-Rechtecks in horizontaler und vertikaler Richtung an.



Durch die zweite Syntax-Variante ist es möglich, im Koordinaten-Paar ***'Xre'*** und ***'Yun'*** hinter ***TO*** die absolute Position der rechten, unteren Ecke des CLIP-Rechtecks anzugeben.

Die dritte Syntax-Variante bewirkt eine Beschränkung von Grafikausgaben auf den Arbeitsbereich eines durch **OPENW#** geöffneten GFA-Fensters mit der angegebenen ***'#Nummer'***. Die Position und Ausmaße der Arbeitsfläche dieses Fensters werden dann als CLIP-Rechteck installiert.

Die Option ***OFFSET 'X','Y'*** kann an die verschiedenen CLIP-Varianten angehängt werden, wodurch ein neuer Koordinaten-Nullpunkt für die folgenden Grafik-Ausgaben bestimmt wird. Alle Grafik-Ausgaben werden dann um den Betrag ***'X'*** in horizontaler und um ***'Y'*** in vertikaler Richtung versetzt gezeichnet.

Die vierte Variante zeigt, daß die Option ***OFFSET 'X','Y'*** auch allein eingesetzt werden kann. Sie legt dann ebenfalls - wie eben beschrieben - den neuen Koordinaten-Nullpunkt fest.

CLIP OFF { CLI O } Grafik-Clipping aufheben

```
CLIP OFF
```

CLIPP OFF schaltet das aktuelle Clipping (s. **CLIP**) wieder aus, sodaß wieder der gesamte Bildschirm für die Grafik-Ausgabe zur Verfügung steht.

GETSIZE() Speicherbedarf für GET-Befehl ermitteln

```
Var=GETSIZE(X_links,Y_oben,X_rechts,Y_unten)
```

Mit dieser Funktion kann vor Ausführung eines **GET**-Befehls im Grafikmodus ermittelt werden, wieviel Speicherplatz der durch die beiden Koordinatenpaare ***'X_links','Y_oben'*** und ***'X_rechts', 'Y_unten'*** beschriebene Ausschnitt bei einer Speicherung mittels GET verbrauchen würde. In den Fällen, daß **GET** mehr als 32767 Bytes benötigen würde, kann dadurch die Fehlermeldung *'String zu lang'* vermieden werden.

GET { GE } Bildschirmbereich im Grafikmodus speichern

```
GET X_links,Y_oben,X_rechts,Y_unten,Var$
```

Durch die Koordinatenpaare ***'X_links'***, ***'Y_oben'*** und ***'X_rechts'***, ***'Y_unten'*** wird im Grafikmodus ein Bildschirmausschnitt definiert, welcher als Bitmuster in die Stringvariable ***'Var$'*** eingelesen wird und dann durch **PUT** wieder an eine beliebige Bildschirmposition zurückkopiert werden kann. Überschreitet der Speicherbedarf des Ausschnitts die max. Stringlänge von 32767 Byte, so muß der Ausschnitt entsprechend reduziert oder in kleinere Abschnitte unterteilt werden, die dann getrennt durch GET gesichert werden. Um die Fehlermeldung *'String zu lang'* zu vermeiden, kann durch die Funktion **GETSIZE()** der Speicherbedarf eines Ausschnitts vorher ermittelt werden.

POINT() Farbwert eines Bildpunktes ermitteln

```
Var=POINT(Xpos,Ypos)
```

Liefert im Grafikmodus die Nummer des Farbregisters, aus dem der durch ***'Xpos'***, ***'Ypos'*** bezeichnete Bildschirm-Punkt seine Farbe bezieht (s. **COLOR**).

PUT { PU } Bildschirmbereich im Grafikmodus setzen

```
PUT X_links,Y_oben,Var$ [,Modus]
```

Zeichnet im Grafikmodus einen durch **GET** eingelesenen Bildausschnitt an die Koordinaten ***'X_links'/'Y_oben'***. Die ursprüngliche Größe bleibt dabei unverändert.

Durch die Option ***'Modus'*** kann bestimmt werden, in welcher Art der zu zeichnende Ausschnitt mit dem bereits bestehende Hintergrund verknüpft wird. Dabei gelten dieselben Festlegungen, wie unter **GRAPHMODE** beschrieben:

'Modus':

1	=	replace
2	=	OR (transparent)
3	=	XOR
4	=	NOT AND (invers transparent)

Wird ***'Modus'*** nicht angegeben, wird im *Replace*-Modus gezeichnet.

RC_INTERSECT() Überlappung zweier Rechtecke ermitteln

```
Var=RC_INTERSECT(X1,Y1,B1,H1,X2&,Y2&,B2&,H2&)
```

Diese Funktion stellt fest, ob sich die beiden angegeben Rechtecke in einem Bereich überschneiden (z.B. für *Window-Redraw* sehr wichtig).

Das erste Rechteck wird durch Angabe des Koordinaten-Paars ***'X1'*** und ***'Y1'*** für die linke, obere Ecke, sowie durch seine Breite in ***'B1'*** und seine Höhe in ***'H1'*** beschrieben. Die Angabe dieser Parameter kann beliebig erfolgen (num. Expr., num.Var., Wert). Die Koordinaten und Maße des zweiten Rechtecks sind dagegen in Integervariablen zu übergeben, da diese bei Abschluß der Funktion als Rückgabevariablen benötigt werden.

Bei Überschneidung der beiden Rechtecke enthält ***'Var'*** TRUE (-1) und in den Variablen ***'X2&', 'Y2&', 'B2&'*** und ***'H2&'*** werden die Koordinaten und Maße der Schnittfläche zurückgegeben. Überschneiden sich die Rechtecke nicht, enthält ***'Var'*** FALSE (0) und in ***'X2&', 'Y2&', 'B2&'*** und ***'H2&'*** werden die Koordinaten und Maße des dazwischenliegenden Rechtecks geliefert. Die zurückgegebene Breite oder/und Höhe ist/sind in diesem Fall negativ.

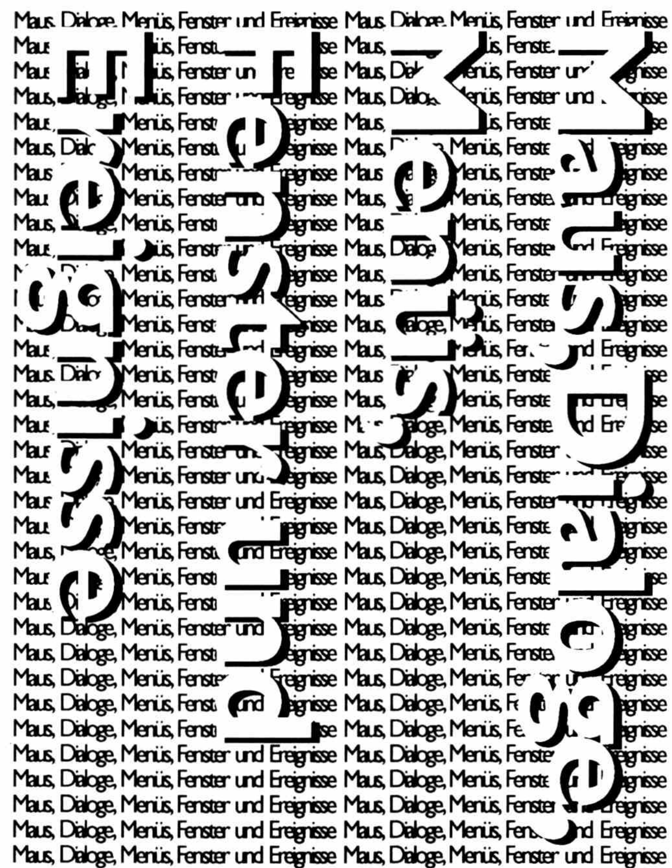
SCREEN { sc } Bildschirm-Modus bestimmen

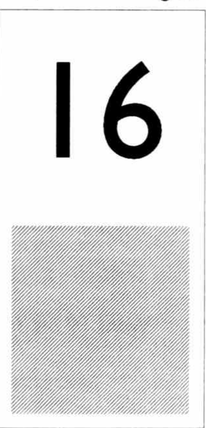
```
SCREEN Modus
```

Durch SCREEN können - abhängig von den Fähigkeiten des angeschlossenen Grafik-Adapters - verschiedene Grafikmodi eingestellt werden. (Tabelle s. Folgeseite).

_MD		_C	_ADAP	Karte	_X	_Y	_TS
Modus	**Typ**	**Farbe**			**Breite**	**Höhe**	**Adresse**
-1	Grafik	2	1	HGC	720	348	$B000
0	Text	16	2	CGA	40	25	$B800
1	Text	16	2	CGA	40	25	$B800
2	Text	16	2	CGA	80	25	$B800
3	Text	16	2	CGA	80	25	$B800
4	Grafik	4	2	CGA	320	200	$B800
5	Grafik	4	2	CGA	320	200	$B800
6	Grafik	2	2	CGA	640	200	$B800
7	Text	2	0	MDA/HGC	80	25	$B000
8-12	------	--	--	----	---	---	-----
13	Grafik	16	3/4	EGA/VGA	320	200	$A000
14	Grafik	16	3/4	EGA/VGA	640	200	$A000
15	Grafik	2	3/4	EGA/VGA	640	350	$A000
16	Grafik	16	3/4	EGA/VGA	640	350	$A000
17	Grafik	2	4	VGA	640	480	$A000
18	Grafik	16	4	VGA	640	480	$A000
19	Grafik	256	4	VGA	320	200	$A000

Notizen:


Maus, Dialoge, Menüs, Fenster und Ereignisse


16

16. MAUS, DIALOGE, MENÜS, FENSTER UND EREIGNISSE

16.1. MAUS - BEFEHLE

DEFMOUSE { DEFMO } Mausform bestimmen

```
DEFMOUSE Form
DEFMOUSE Adr%
```

DEFMOUSE ermöglicht im Grafikmodus den Einsatz verschiedener Mausformen.

'Form':

0 = Pfeil
1 = X-Klammer (Text-Cursor)
2 = Biene
3 = zeigende Hand
4 = offene Hand
5 = Fadenkreuz fein
6 = Fadenkreuz grob
7 = Fadenkreuz umrandet

Mit der zweiten Syntaxform ist es möglich, einen eigenen Mauszeiger zu definieren, bzw. zu installieren. Dazu wird dem DEFMOUSE-Befehl in ***'Adr%'*** die Startadresse eines 34 Words (68 Bytes) langen Mausdefinitions-Strings übergeben.

In diesem String sind der Reihe nach folgende Daten vorzubereiten:

Word 1	=	X-Koordinate des Aktionspunktes
Word 2	=	Y-Koordinate des Aktionspunktes
Word 3 bis 18	=	16 Bitmuster-Words, die - von oben nach unten laufend - die 16 Pixelzeilen des Mausmusters (schwarz) darstellen
Word 19 bis 34	=	16 Bitmuster-Words, die - von oben nach unten laufend - die 16 Pixelzeilen der Mausmaske (weiß) darstellen

Die Bit-Kombination von Maske und Muster (je 1 Bit übereinanderliegend gedacht) ergibt 4 verschiedene Verknüpfungsmöglichkeiten der Maus mit dem Hintergrund:

Muster (schwarz)	**Maske** (weiß)	**Maus-Pixel** wird
0	0	transp.
1	0	schwarz
0	1	XOR
1	1	weiß

Beispiel:

```
SCREEN 18            // Farb-VGA an (auch mit EGA/CGA)
BOX 100,100,200,200         // Aktionsbox zeichnen
'
maus1$=MKI$(8)+MKI$(8)      // Aktionspunkt für Maus 1
maus2$=MKI$(0)+MKI$(0)      // Aktionspunkt für Maus 2
FOR i|=0 TO 15                 // 16 Zeilen
  READ bin1$,bin2$             // je Bitmuster lesen
  muster$=muster$+MKI$(VAL(bin1$)) // Vordergrund
  maske$=maske$+MKI$(VAL(bin2$))   // Hintergrund
NEXT i|                        // nächste Zeile
maus1$=maus1$+muster$+maske$ // Maus1: Muster dann Maske
maus2$=maus2$+maske$+muster$ // Maus2: umgekehrt
'
DEFMOUSE V:maus1$             // Maus 1 einschalten
PRINT ''Abbruch   : beide Maustasten''
PRINT ''Zeichnen  : linke Maustaste''
PRINT ''Pause     : rechte Maustaste''
REPEAT                        // grosse Demoschleife
IF MOUSEK=1 THEN PCIRCLE MOUSEX,MOUSEY,3 // zeichnen,
'                             // wenn linke Maustaste
IF MOUSEK=2                   // wenn rechte Maustaste
  REPEAT                      // Warteschleife
    k%=MOUSEK
  UNTIL K%=0 OR K%=3          // bis Taste gelöst oder
ENDIF                         // Abbruch
IF @mbound(100,100,200,200)// Maus innerhalb der Box?
  flag!=flag! XOR TRUE        // Maus-Flag umschalten
  COLOR RAND(15)+1            // Farbe variieren
  TEXT 0,64,''Mausform  : ''+STR$(ABS(flag!)+1)
  PBOX 100,100,200,200        // neue Box zeichnen
  IF flag!=0                  // Flag aus?
    TEXT 0,80,''Aktionspkt: zentriert (8/8)''
    DEFMOUSE V:maus1$         // dann Maus 1 wählen
  ELSE                        // Flag an!
    TEXT 0,80,''Aktionspkt: linksoben (0/0)''
    DEFMOUSE V:maus2$         // dann Maus 2 wählen
  ENDIF
  SHOWM                       // Maus anschalten
  PAUSE 10                    // kleine Pause
```

```
  ENDIF
UNTIL MOUSEK=3          // Exit, wenn beide Maustasten
'
'       Muster-Datas        Masken-Datas
DATA %1111111111111111,%1111111111111111
DATA %1111111111111111,%1000000000000001
DATA %1100000000000011,%1011111111111101
DATA %1100000000000011,%1011111111111101
DATA %1100111111110011,%1011000000001101
DATA %1100100000010011,%1011000000001101
DATA %1100100000010011,%1011000000001101
DATA %1100100000010011,%1011000000001101
DATA %1100100000010011,%1011000000001101
DATA %1100100000010011,%1011000000001101
DATA %1100100000010011,%1011000000001101
DATA %1100111111110011,%1011000000001101
DATA %1100000000000011,%1011111111111101
DATA %1100000000000011,%1011111111111101
DATA %1111111111111111,%1000000000000001
DATA %1111111111111111,%1111111111111111
'
FUNCTION mbound(xl,yo,xr,yu)
  ' Diese nützliche Routine überprüft die aktuelle
  ' Mausposition darauf, ob sie sich innerhalb des
  ' angegebenen Bildschirmbereichs befindet.
  LOCAL X&,Y&,k&            // Lokal-Deklar.
  MOUSE X&,Y&,k&            // Mausstatus ermitteln
  IF X&=>xl AND X&<=xr AND Y&=>yo AND Y&<=yu...
                            ...THEN RETURN TRUE
  RETURN FALSE              // Rückgabe=0,...
ENDFUNC                     // ... wenn Maus außerhalb
```

HIDEM { HI } — Mauszeiger ausschalten

```
HIDEM
```

Der Mauszeiger wird ausgeschaltet, die Koordinatenermittlung mittels der **MOUSE**-Befehle funktioniert jedoch auch weiterhin. Bei **FILESELECT-**, **POPUP-** und **ALERT**-Aufrufen wird intern automatisch **SHOWM** ausgeführt. Nach der Bedienung solcher Dialog-Boxen muß gfls. erneut HIDEM ausgeführt werden..

MOUSE { MO } — Maus- und Shift-Status gesamt ermitteln

```
MOUSE Xvar&,Yvar&,Bvar& [,Shiftvar&]
```

In den Integer-Rückgabe-Variablen ***'Xvar&'*** und ***'Yvar&'*** wird die aktuelle X-, bzw.Y-Koordinate des Mauszeigers, sowie in ***'Bvar&'*** der Status der Maustasten geliefert:

0	=	keine Taste gedrückt
1	=	linke Taste gedrückt
2	=	rechte Taste gedrückt
3	=	linke und rechte Taste gedrückt

Für *'Drei-Tasten-Mäuse'*:

4	=	mittlere Taste gedrückt
5	=	linke und mittlere Taste gedrückt
6	=	rechte und mittlere Taste gedrückt
7	=	alle drei Tasten sind gedrückt

Im aktiven Textmodus werden für die Koordinatenberechnung die Cursorspalte (Standard: 1 - 80) und Cursorzeile (Standard: 1 bis 25), in welcher sich die Maus momentan befindet, mit dem Faktor 8 multipliziert und davon der Wert 8 abgezogen.

Bei geöffneten GFA-Fenstern bzw. bei aktivem **CLIP OFFSET** werden in ***'Xvar&'*** und/oder ***'Yvar&'*** auch negative Werte geliefert, wenn sich der Mauszeiger links und/oder oberhalb des Fensters bzw. des **CLIP**-Nullpunktes befindet.

Zusätzlich kann durch die optionale Angabe einer weiteren Rückgabe-Variable ***'Shiftvar&'*** der aktuelle Status der *'Shift'*-Sondertasten (Umschalttasten) erfragt werden. Die Tabelle der möglichen Zustände finden Sie hierzu unter **ON MENU KEY GOSUB...**

MOUSEX — Maus-X-Koordinate ermitteln
MOUSEY — Maus-Y-Koordinate ermitteln
MOUSEK — Maustasten-Status ermitteln

```
Var=MOUSEX        ->    aktuelle X-Position
Var=MOUSEY        ->    aktuelle Y-Position
Var=MOUSEK        ->    Maustasten-Status
```

Dieses sind reservierte Variablen, welche den jeweils gewünschten Maus-Status separat enthalten. Weitere Informationen finden Sie unter **MOUSE....**

SHOWM { SHOW } Mauszeiger anschalten

```
SHOWM
```

Der gfls. durch **HIDEM** unsichtbar gemachte Mauszeiger wird wieder eingeschaltet. Zum Programmende wird intern immer SHOWM ausgeführt.

16.2. DIALOG - BEFEHLE

ALERT { AL } Hinweis-Box erzeugen

```
ALERT,Icon,Bx_text$,Button,Bu_text$,Var
```

Der ALERT-Befehl produziert je nach **SCREEN**-Modus eine mittig plazierte Hinweisbox, die mit einem beliebigen Text und mit bis zu drei Auswahlknöpfen (*'Klickfelder'* oder engl.: *'buttons'*) ausgestattet werden können. Die Programm-Ausführung wird dabei solange angehalten, bis der Anwender entweder durch Maus- oder durch Tastaturbedienung einen der Buttons ausgewählt hat.

Icon:
0 = kein Symbol
1 = Ausrufungszeichen
2 = Fragezeichen
3 = STOP-Schild

Bx_text$:

Hier wird der eigentliche Text (Mitteilung/Frage) an die Funktion übergeben. Das Zeichen "|" (*Pipe*) gilt darin als Trennungszeichen zwischen den einzelnen Zeilen. Die maximal mögliche Anzahl an Zeilen und die maximale Zeilenlänge ist von dem jeweils eingestellten **SCREEN**-Modus abhängig. Sollten zuviel Zeilen angegeben worden sein oder eine der Zeilen überschreitet die jeweils mögliche Länge, so wird keine ALERT-Box gezeichnet. Der Text kann direkt in den Befehl geschrieben, oder auch als Stringvariable übergeben werden.

Button:

Es wird die Nummer (1 bis *x*) des Buttons übergeben, welcher ausser durch Mausklick auch durch die <Return>-Taste (*default*) bestätigt werden kann. Dieser Button wird in der Box stark umrandet gezeichnet. *'x'* hängt auch hier

von der maximal möglichen Button-Anzahl ab (0 = kein Default-Button). Durch Betätigung der horizontalen Scroll-<Pfeiltasten> kann der Defaultbutton auch noch während der Boxbedienung bestimmt werden.

Bu_text$:

Durch diesen Parameter erfolgt die Beschriftung der einzelnen Buttons. Die Länge des längsten Buttontextes bestimmt die einheitliche Breite aller Buttons. Dabei ist die maximale Buttontextlänge wiederum von dem aktuellen **SCREEN**-Modus und von der Anzahl der angegebenen Buttons abhängig. Wird die gesamte Buttondarstellung zu lang, wird keine ALERT-Box gezeichnet. Auch hier gilt das *Pipe-Zeichen* ("|") als Trennstrich zwischen den einzelnen Buttontexten.

Zusätzlich kann in den einzelnen Buttontexten durch einen Tiefstrich vor einem Buchstaben bestimmt werden, durch welche Taste dieser Button auch ohne Maus gewählt werden kann. Z.B. `''E_XIT''` bewirkt, daß der Button auch durch Druck auf die <X>-Taste gewählt werden kann. Der Unterstrich bleibt bei der Ausgabe des Buttontextes unberücksichtigt. Der Text kann - wie bei ***'Boxtext$'*** - direkt angegeben oder in einer Stringvariablen übergeben werden.

Var:

Dies ist eine numerische Rückgabe-Variable, in welcher der Befehl die Nummer des vom Anwender gewählten Buttons (1 bis x) zurückgibt.

FILESELECT { FILESE } Datei auswählen

```
FILESELECT Pfad$,Auswahl$,Backvar$
```

Dieser Befehl erstellt ein Dialog-Formular zur Dateiauswahl im *SAA*-Standard, ermöglicht eine Dateiauswahl per Maus oder/und Tastatur und liefert gfls. den gewählten Dateinamen (gfls. incl. Pfad).

'Pfad$'*** bezeichnet den Suchpfad zur gesuchten Datei. Durch Angabe des Suchpfades `'A:\ROUTINEN\*.LST'` kann z.B. erreicht werden, daß nur aus dem Verzeichnis *'Routinen'* auf Laufwerk *'A:'* alle Dateien angezeigt werden, die die Extension *'.LST'* besitzen. Wird kein ***'Pfad$' übergeben (" "), werden alle Dateien des aktuellen Pfades aufgelistet.

Der Parameter ***'Auswahl$'*** gibt gfls. einen Dateinamen (max. 12 Zeichen incl.Trennpunkt und Extension) an, der bei Aufruf der Box in der Eintragszeile voreingestellt werden soll. ***'Pfad$'*** und ***'Auswahl$'*** können auch als zusammengesetzter String-Ausdruck oder als Text direkt übergeben werden.

Die Stringvariable ***'Backvar$'*** enthält nach Abschluß der Datei-Auswahl den Namen (gfls. incl. Pfad) der gewählten Datei. Hierbei sind vier verschiedene Eintragsvarianten möglich:

- wurde vom Anwender eine Datei gewählt, steht ihr Name anschließend vollständig in der Rückgabevariablen ***'Backvar$'***.
- wurde ohne Auswahl die *'OK-Box'* bedient, gibt es zwei Varianten:
  - es wurde durch den Parameter ***'Auswahl'*** ein Dateiname in der Eintragszeile voreingestellt, der noch in der Eintragszeile steht und die *'Ok-Box'* wurde bedient. Dann steht dieser Name auch anschließend in ***'Backvar$'***.
  - es wurde keine Auswahl getroffen und auch kein Name übergeben, bzw. die Eintragszeile wurde vom Anwender gelöscht und die *'OK-Box'* wurde bedient, dann wird in ***'Backvar$'*** ein sog. *'Backslash'* (\) geliefert.
- Wird *'Abbruch'* angeklickt, ist ***'Backvar$'*** absolut leer.

Die Boxbedienung per Maus ist relativ einfach und logisch, sodaß ich mir hier weitere Erläuterungen erspare. Bei Tastaturbedienung kann per <Tab>-Taste durch die verschiedenen Eintrags- und Auswahlfelder *'gesprungen'* werden. Innerhalb des Dateiauswahl-Fensters können die jeweiligen Einträge durch die vertikalen Scroll-<Pfeiltasten> ausgewählt und anschließend durch <Return> in die Eintragszeile übertragen werden.

POPUP() — Pop-Up-Menü erzeugen

```
Auswahl=POPUP(Menütext$,Xpos,Ypos,Modus)
~POPUP(Menütext$,Xpos,Ypos,Modus)
```



POPUP wird sicher auch einer Ihrer Lieblingsbefehle. Er hört sich an wie *'POPEYE'* und ist auch ebenso vielseitig. Man kann damit ein (fast) beliebig großes Pop-Up-Menü produzieren und an einer beliebigen Bildschirmposition darstellen. Der Programmablauf

wird für die Zeit der Menü-Bedienung unterbrochen und man kann anhand der Cursor-<Pfeiltasten>, der Maus und/oder der <Return>-Taste die Auswahl vornehmen. Ein Mausklick außerhalb des Menüs gilt als *'Abbruch'*.

In ***'Menütext$'*** werden - durch ein *'Pipe'-Zeichen* (|) getrennt - die einzelnen Menüzeilen-Einträge (mind. 3) angegeben. Der Menüzeilen-Text vor dem ersten *'Pipe'* wird als unwählbarer Menütitel invertiert dargestellt, während alle anderen Einträge normal dargestellt werden und wählbar sind. Soll kein Menütitel gezeichnet werden, so steht gleich zu Beginn in ***'Menütext$'*** ein *'Pipe'-Zeichen* (|), der Menütitel-Text wird also weggelassen. Dadurch wird erreicht, daß schon die erste dargestellte Menüzeile wählbar ist (wichtig für den *Pulldown-Modus*: ***'Modus'***=3).

Wie auch bei **ALERT** ist die maximale Anzahl und Länge vom aktuellen **SCREEN**-Modus abhängig. Sollten zuviel Zeilen angegeben worden sein oder eine der Menüzeilen ist zu lang, so wird das Menü nicht gezeichnet und der Wert -1 (**TRUE**) zurückgegeben. Ebenfalls wie bei **ALERT** kann jedem Menü-Eintrag ein sog. *'Hotkey'* zugewiesen werden, indem dem betreffenden Zeichen ein Tiefstrichs vorangestellt wird. Durch Druck auf die entsprechende <Taste> kann der jeweilige Eintrag auch ohne Zuhilfenahme einer Maus angewählt werden.

Das Koordinatenpaar ***'Xpos'*** und ***'Ypos'*** gibt entweder die linke, obere Ecke (***'Modus'***=0) oder die Mitte des Menüs an (***'Modus'***=1). Bei ***'Modus'***=2 und ***'Modus'***=3 werden die evtl. in ***'Xpos'*** und ***'Ypos'*** übergebenen Werte ignoriert.

'Modus' entscheidet über die Arbeitsweise des Menüs:

0 = Die in ***'Xpos'/'Ypos'*** angegebenen Koordinaten werden als linke, obere Ecke des Menüs gewertet.

1 = Die in ***'Xpos'/'Ypos'*** angegebenen Koordinaten werden als Mitte des Menüs gewertet.

2 = Die in ***'Xpos'/'Ypos'*** angegebenen Koordinaten werden garnicht gewertet. Das Menü wird in der Bildschirmmitte zentriert ausgegeben.

3 = *Pulldown-Modus*. Die in ***'Xpos'/'Ypos'*** angegebenen Koordinaten werden als linke, obere Ecke des Menüs gewertet. Anders als bei ***'Modus'***=0 kann hier das Menü auch durch die horizontalen Scroll-<Pfeiltasten> nach links und rechts verlassen werden. Dabei wird als Rückgabewert

-3 = <Pfeiltaste-links>)
-4 = <Pfeiltaste-rechts>

geliefert.

Außerdem wird bei einem Klick der linken Maustaste ausserhalb des Menüs der Wert -2 als Kennung geliefert. Soll das Menü ohne Titelzeile dargestellt werden, muß ***'Menütext$'*** einem *'Pipe'-Zeichen* beginnen.

In der Rückgabe-Variable ***'Auswahl'*** wird bei getätigter Auswahl die Nummer des gewählten Eintrags - beginnend mit 1 für den ersten wählbaren Eintrag - geliefert.

Die zweite Syntax-Variante zeigt die Möglichkeit, anstatt eines Rückgabewertes einen Eintrag in den **MENU()**-Ereignispuffer zu erzwingen. Durch **GETEVENT, PEEKEVENT** oder **ON MENU** (s. dort) wird der Puffer aktualisiert und es kann anschließend festgestellt werden, ob ein Eintrag eines POPUP-Menüs gewählt wurde. **MENU(1)** liefert dann den Wert 20. In **MENU(0)** steht gfls. anschließend - wie sonst in der Rückgabevariable ***'Auswahl'*** - die Nummer des gewählten Eintrags.

DRAGBOX — Schiebebox produzieren

```
DRAGBOX Sx,Sy,Br,Ho [,Mx,My,Mb,Mh] ,Ex&,Ey&
```

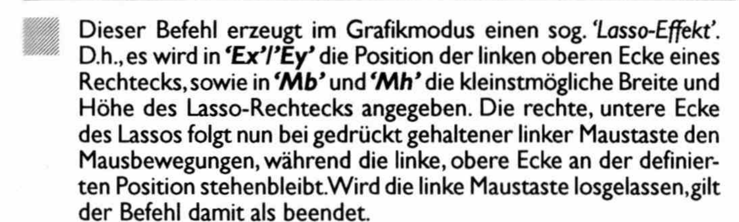
Erzeugt im Grafikmodus eine *Schiebebox*, die in der vorgegebenen Größe - ***'Br'*** für Breite, ***'Ho'*** für Höhe, beginnend mit der linken, oberen Boxecke bei der Startkoordinate ***'Sx'/'Sy'*** - mittels Maus beliebig auf dem Bildschirm verschoben werden kann. Innerhalb von Fenstern wird die Boxdarstellung allerdings an den Rändern der Arbeitsfläche ge*'clipt'*.

Der Befehl ist nur bei gedrückter linker Maustaste aufrufbar. Solange die Maustaste gedrückt ist, folgt das Rechteck dem Mauszeiger. Wird die Taste losgelassen, verschwindet die Box und in den Integer-Rückgabe-Variablen ***'Ex&'*** und ***'Ey&'*** stehen die Koordinaten der linken, oberen Ecke der Schiebebox zum Zeitpunkt des Loslassens.

Werden die optionalen Parameter ***'Mx'***, ***'My'***, ***'Mb'*** und ***'Mh'*** eingesetzt, so beschreiben sie in ***'Mx'/'My'*** die linke, obere Ecke und in ***'Mb'/'Mh'*** die Breite und Höhe eines Begrenzungsrechtecks. Die bewegliche Box läßt sich dann nicht mehr über die vorgegebenen Grenzen dieses unsichtbaren Rahmens hinausbewegen.

RUBBERBOX Gummiband-Box (Lasso) produzieren

```
RUBBERBOX Sx,Sy,Mb,Mh,Ex&,Ey&
```



Dieser Befehl erzeugt im Grafikmodus einen sog. *'Lasso-Effekt'*. D.h., es wird in ***'Ex'/'Ey'*** die Position der linken oberen Ecke eines Rechtecks, sowie in ***'Mb'*** und ***'Mh'*** die kleinstmögliche Breite und Höhe des Lasso-Rechtecks angegeben. Die rechte, untere Ecke des Lassos folgt nun bei gedrückt gehaltener linker Maustaste den Mausbewegungen, während die linke, obere Ecke an der definierten Position stehenbleibt. Wird die linke Maustaste losgelassen, gilt der Befehl damit als beendet.

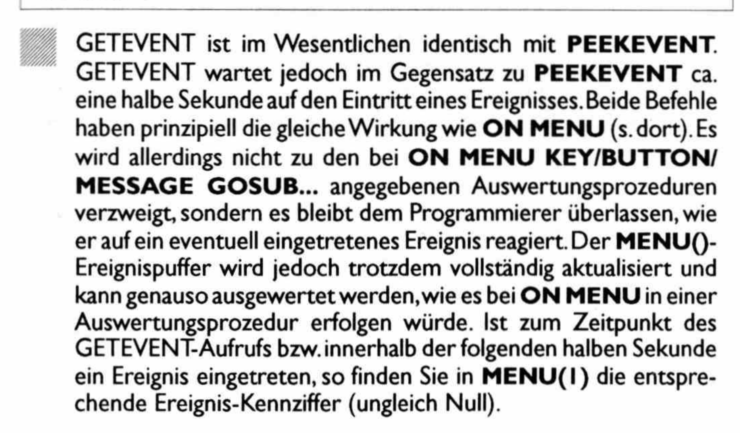
Daraus folgt, daß RUBBERBOX nur bei gedrückt gehaltener linker Maustaste aufgerufen werden kann. Als Rückgabeparameter erhält man in den beiden Integer-Rückgabe-Variablen ***'Ex'*** und ***'Ey'*** nach Abschluß die Breite und Höhe des Lasso-Rechtecks zum Zeitpunkt des Loslassens der Maustaste.

Werden in ***'Mb'*** und ***'Mh'*** negative Werte angegeben, kann die Box auch nach links und oben *'gezogen'* werden.

16.3. EREIGNIS - ÜBERWACHUNG

GETEVENT **{ GETE }** wartende Event-Kontrolle ohne Verzwg.

```
GETEVENT
```



GETEVENT ist im Wesentlichen identisch mit **PEEKEVENT**. GETEVENT wartet jedoch im Gegensatz zu **PEEKEVENT** ca. eine halbe Sekunde auf den Eintritt eines Ereignisses. Beide Befehle haben prinzipiell die gleiche Wirkung wie **ON MENU** (s. dort). Es wird allerdings nicht zu den bei **ON MENU KEY/BUTTON/ MESSAGE GOSUB...** angegebenen Auswertungsprozeduren verzweigt, sondern es bleibt dem Programmierer überlassen, wie er auf ein eventuell eingetretenes Ereignis reagiert. Der **MENU()**-Ereignispuffer wird jedoch trotzdem vollständig aktualisiert und kann genauso ausgewertet werden, wie es bei **ON MENU** in einer Auswertungsprozedur erfolgen würde. Ist zum Zeitpunkt des GETEVENT-Aufrufs bzw. innerhalb der folgenden halben Sekunde ein Ereignis eingetreten, so finden Sie in **MENU(1)** die entsprechende Ereignis-Kennziffer (ungleich Null).

KILLEVENT — MENU()-Ereignispuffer löschen

```
KILLEVENT
```



In manchen Fällen ist es denkbar, daß eintretende Ereignisse im Voraus feststehen und der Eintrag im ***MENU()***-Ereignispuffer daher überflüssig wird (z.B. beim Aufbau der Rechtecklisten während der erstmaligen Installation eines Fensters). In diesen Fällen oder evtl. vor einem **GETEVENT**-Aufruf zur *'Lauerstellung'* kann durch KILLEVENT der **MENU()**-Ereignispuffer vollständig gelöscht werden.

ON MENU — Event-Kontrolle mit Verzweigung

```
ON MENU
```



Dies ist die zentrale *Multi-Event-Funktion* des GFA-BASIC. Sie stellt fest, ob eines der vielfachen Multi-Ereignisse (Maus- oder Tastatur-Klicks, Menü- oder/und Fenster-Ereignisse usw.) eingetreten ist, aktualisiert entsprechend die Einträge im **MENU()**-*Ereignispuffer* (s. dort) und verzweigt gfls. entsprechend zu den - vorher durch die **ON MENU...GOSUB**-Befehle angegebenen - Verwaltungs-prozeduren.

Die 'ON MENU'-Ereignisfeststellung sollte an häufig wieder-kehrenden Programmstellen (innerhalb von Schleifen, möglichst in der Hauptschleife in möglichst schneller Folge) geschehen, da der **MENU()**-*Ereignispuffer* permanent erneuert werden muß, um gfls. schnellstmöglich auf ein eingetretenes Ereignis reagieren zu können. Geschieht dies nämlich nicht unverzüglich und der Puffer wird nicht ausgelesen, können neuere Ereignisse die Puffer-Einträge überschreiben und somit ein falsches Ergebnis liefern, bzw. das betreffende Ereignis wird dann möglicherweise schlicht und einfach *'übersehen'*.

Wird der Befehl nicht eingesetzt, kann auch nicht zu den gfls. bei den **ON MENU...GOSUB**-Befehlen angegebenen Prozeduren verzweigt werden.

Beachten Sie dazu bitte das ausführliche 'BEISPIEL - PROGRAMM' im ANHANG.

PEEKEVENT { PEEKE } Event-Kontrolle ohne Verzweigung

```
PEEKEVENT
```

PEEKEVENT ist im Wesentlichen identisch mit **GETEVENT** (s. Erläuterungen auch dort). PEEKEVENT wartet im Gegensatz zu **GETEVENT** nicht auf den Eintritt eines Ereignisses. Beide Befehle haben prinzipiell die gleiche Wirkung wie **ON MENU** (weitere Erläuterungen siehe dort).

16.4. EREIGNIS-VERWALTUNG

MENU() allgemeiner Ereignispuffer

```
Var=MENU(Index)
```



Hinter dieser Funktion verbirgt sich ein Integer-Vektor, in welchen durch **ON MENU, PEEKEVENT** oder **GETEVENT** der permanente Eintrag verschiedener Daten zu den aktuellen GFA-Multi-Ereignissen erzwungen werden kann.

'Index' steht hier jeweils für das Puffer-Element, das ausgelesen werden soll. Das wichtigste Puffer-Element ist MENU(1), da hier die Kennziffer des jeweils eingetretenen Ereignisses zu finden ist. Abhängig davon stehen dann in anderen Puffer-Elementen weitere Auskünfte zu dem Ereignis (hier durch `'->'` markiert).

MENU(1):	**Ereignis:**	
1	Tastatur-Ereignis	
	-> **MENU(5)**	16Bit-Tastaturcode (s. ANHANG 'PC-TASTATUR')
2	Mausklick außerhalb des aktiven Fensters	
	-> **MENU(7)**	gfls. Nummer des Fensters, auf welchem sich die Maus zum Klickzeitpunkt befunden hat. War kein Fenster darunter, wird eine Null geliefert.
3	Mausklick innerhalb des aktiven Fensters	

4	Der Schließbutton *'Closer'* des Fensters links oben wurde gewählt.	
5	Der Größen-Minimierungsbutton *'Minimizer'* (Abwärtspfeil) rechts oben wurde gewählt	
6	Der Größen-Maximierungsbutton *'Maximizer'* (Aufwärtspfeil) rechts oben wurde gewählt	
7	Der Aufwärts-Zeilen-Scrollpfeil rechts über dem vertikalen Schieberegler wurde gewählt	
8	Der Abwärts-Zeilen-Scrollpfeil rechts unter dem vertikalen Schieberegler wurde gewählt	
9	Der Spalten-Scrollpfeil links neben dem horizontalen Schieberegler wurde gewählt	
10	Der Spalten-Scrollpfeil rechts neben dem horizontalen Schieberegler wurde gewählt	
11	Das *'PageUp'*-Feld des vertikalen Scroll-Balkens oberhalb des Schiebereglers wurde gewählt	
12	Das *'PageDown'*-Feld des vertikalen Scroll-Balkens unterhalb des Schiebereglers wurde gewählt	
13	Das *'PageLeft'*-Feld des horizontalen Scroll-Balkens links vom Schieberegler wurde gewählt	
14	Das *'PageRight'*-Feld des horizontalen Scroll-Balkens rechts vom Schieberegler wurde gewählt	
15	Der 'UpDown'-Schieberegler im vertikalen Scroll-Balken wurde bewegt	
	-> **MENU(7)**	neue Position im Scrollbalken von 0 (oben) bis 1000 (unten)
16	Der 'LeftRight'-Schieberegler im horizontalen Scroll-Balken wurde bewegt	
	-> **MENU(7)**	neue Position im Scrollbalken von 0 (links) bis 1000 (rechts)
17	Der Titelbalken *'Mover'* am oberen Rand des aktiven Fensters wurde gewählt.	



```
               |------------------+---------------------------------
               | -> MENU(7)  | gfls. neue X-Position der linken,
               |             | oberen Ecke des Fensters
               | ->  MENU(8) | gfls. neue Y-Position der linken,
               |             | oberen Ecke des Fensters
-------------- +---------------- +--------------------------------
18             | Das Größenfeld 'Sizer' in der rechten, unteren
               | Ecke des aktiven Fensters wurde gewählt.
               |----------------- +-------------------------------
               | -> MENU(7)  | gfls. neue Breite des Fensters
               | -> MENU(8)  | gfls. neue Höhe des Fensters
-------------- +---------------- +--------------------------------
19             | Die Informationszeile 'Info' unterhalb des
               | Titelbalkens am oberen Rand des aktiven
               | Fensters wurde gewählt.
-------------- +-------------------------------------------------
20             | Ein Menü-Eintrag (gfls. auch POPUP-Eintrag,
               | falls  ~POPUP  verwendet wurde) wurde gewählt.
-------------- +-------------------------------------------------
               | -> MENU(0)  | liefert bei Pulldown-Menüs
               |             | den Element-Index des gewählten
               |             | Menü-Punktes im dazugehörigen
               |             | Menütextfeld. Bei POPUP-Menüs
               |             | wird die Nummer der gewählten
               |             | POPUP-Menüzeile geliefert.
               |  -> MENU(7) | liefert die Nummer des geöff-
               |             | neten Pulldown-Menüs
               |             | (links mit 1 beginnend)
               | -> MENU(8)  | liefert die Nummer des gewählten
               |             | Menüpunktes im geöffneten Menü
               |             | (oben mit 1 beginnend)
-------------- +---------------- +--------------------------------
21             | 'Redraw'-Ereignis. Aufgrund des Bewegens, der
               | Größenänderung oder des Schließens des aktiven
               | Fensters bzw. durch Aktivieren eines neuen
               | Fensters wird das Neuzeichnen (Redraw) eines
               | oder mehrerer Bildschirm-Rechtecke notwendig.
               |-----------------+-------------------------------
               | -> MENU(7)  | X-Position der linken, oberen
               |             | Ecke des neu zu zeichnenden
               |             | Bereichs
               | -> MENU(8)  | Y-Position der linken, oberen
               |             | Ecke des neu zu zeichnenden
               |             | Bereichs
               | -> MENU(9)  | Breite des neu zu zeichnenden
               |             | Bereichs
               | -> MENU(10) | Höhe des neu zu zeichnenden
               |             | Bereichs
-------------- +---------------- +--------------------------------
```

MENU(2)	=	absolute Maus-X-Koordinate zum Zeitpunkt des Ereignisses
MENU(3)	=	absolute Maus-Y-Koordinate zum Zeitpunkt des Ereignisses
MENU(4)	=	Mausknopfstatus zum Zeitpunkt des Ereignisses (s. **MOUSEK**)
MENU(6)	=	Umschalttasten-Status zum Zeitpunkt des Ereignisses (s. **ON MENU KEY GOSUB...**)

Die Puffer-Elemente **MENU(2)** und **MENU(3)** enthalten bei allen Ereignissen die absoluten Mauskoordinaten, **MENU(4)** den Mausknopf-Status und **MENU(6)** den Status der Umschalttasten zum Zeitpunkt des Ereignisses.

Beachten Sie dazu bitte das ausführliche 'BEISPIEL - PROGRAMM' im ANHANG.

ON MENU GOSUB — Proc.-Bestimmung (Menü-Event)

```
ON MENU GOSUB Prozedurname
```

'Prozedurname' gibt eine **PROCEDURE** an, zu welcher verzweigt werden soll, wenn zum Zeitpunkt des **ON MENU**-Befehls (s. dort) ein Pulldown-Menüpunkt (Menüeintrag) angeklickt wurde. Über **MENU(0)** oder über **MENU(7)** und **MENU(8)** kann dann dort der gewählte Menüpunkt ermittelt und dementsprechend reagiert werden (s. **MENU()**-*Ereignispuffer)*.

Außer dieser Verzweigung hat ON MENU GOSUB keine Funktion. Der Befehlsteil ***GOSUB*** kann weggelassen werden, er wird vom Interpreter automatisch ergänzt. Wird ein nicht existierender ***'Prozedurname'*** angegeben, kann dadurch erreicht werden, daß die z.Zt. aktive Menü-Überwachung abgeschaltet wird.

Beachten Sie dazu bitte das ausführliche 'BEISPIEL - PROGRAMM' im ANHANG.

ON MENU BUTTON GOSUB — Proc.-Bestimmung (Mausknopf-Event)

```
ON MENU BUTTON GOSUB Prozedurname
```

Anhand der Ereignis-Überwachung durch **ON MENU** (s. dort) kann festgestellt werden, ob eine Maustaste gedrückt wurde. Das Programm verzweigt in diesem Fall zu der durch ***'Prozedurname'***

angegebenen **PROCEDURE**. Jedoch nur dann, wenn aktuell weder ein Pulldown-Menü noch Fenster-Randelemente bedient werden. In ***'Prozedurname'*** können dann durch den Eintrag in **MENU(1)** zwei mögliche Ereignisse ermittelt werden:

MENU(1)= 2 ein Mausklick außerhalb des aktiven Fensters hat stattgefunden, bzw. ein Mausklick hat stattgefunden und es sind keine Fenster geöffnet. Der Mausklick galt dabei nicht der Bedienung des Pulldown-Menüs. Die Nummer eines evtl. angeklickten Fensters kann gfls. aus **MENU(7)** ausgelesen werden.

MENU(1) = 3 ein Mausklick innerhalb des aktiven Fensters hat stattgefunden. Der Mausklick galt dabei nicht der Bedienung Fenster-Randelemente des aktiven Fensters (s. dazu **ON MENU MESSAGE GOSUB...**)

Beachten Sie die weiteren Erläuterungen zum **MENU()**-*Ereignispuffer*.

Außer dieser Verzweigung hat ON MENU BUTTON GOSUB keine Funktion. Der Befehlsteil ***GOSUB*** kann weggelassen werden, er wird vom Interpreter automatisch ergänzt. Wird ein nicht existierender ***'Prozedurname'*** angegeben, kann dadurch erreicht werden, daß die z.Zt. aktive Maus-Überwachung abgeschaltet wird.

Beachten Sie dazu bitte das ausführliche 'BEISPIEL - PROGRAMM' im ANHANG.

ON MENU KEY GOSUB Proc.-Bestimmung (Tastatur-Event)

```
ON MENU KEY GOSUB Prozedurname
```

Die Tastatur wird durch **ON MENU** (s. dort) überwacht und gfls. bei einem Tastatur-Ereignis zu der in ***'Prozedurname'*** genannten **PROCEDURE** verzweigt. **MENU(1)** liefert dann den *Event-Index 1* (s. **MENU()**-*Ereignispuffer*). Durch

```
BYTE(MENU(5))
```

kann zur gleichen Zeit der ASCII-Code und durch

```
SHR(MENU(5),8)
```

der Scan-Code der gedrückten Taste ermittelt werden (s. dazu ANHANG 'PC-TASTATUR').

MENU(6) liefert dagegen den aktuellen Zustand der Umschalttasten:

Bit0	(1)	=	rechte <Shift>-Taste gedrückt
Bit1	(2)	=	linke <Shift>-Taste gedrückt
Bit2	(4)	=	rechte <Strg>-Taste gedrückt
Bit3	(8)	=	rechte <Alt Gr>-Taste gedrückt
Bit4	(16	=	*'ScrollLock'* ist aktiv
Bit5	(32)	=	*'NumLock'* ist aktiv
Bit6	(64)	=	*'CapsLock'* ist aktiv
Bit7	(128)	=	*'Einfüg'-Modus* ist aktiv
Bit8+Bit4	(260)	=	linke oder beide <Strg>-Tasten
Bit9+Bit3	(520)	=	linke oder beide <Alt>-Taste
Bit12	(4096)	=	<ScrollLock> wird bei aktivem *'ScrollLock'* gedrückt gehalten
Bit12+Bit4	(4112)	=	<ScrollLock> wird vor aktivem *'ScrollLock'* gedrückt gehalten
Bit13	(8192	=	<NumLock> wird bei aktivem *'NumLock'* gedrückt gehalten
Bit13+Bit5	(8224)	=	<NumLock> wird vor aktivem *'NumLock'* gedrückt gehalten
Bit14+Bit6	(16448)	=	<CapsLock> wird gedrückt gehalten
Bit15	(32768)	=	<Einfg> wird bei aktivem *'Einfüg'-Modus* gedrückt
Bit15+Bit7	(32896)	=	<Einfg> wird vor aktivem *'Einfüg'-Modus* gedrückt

Die verschiedenen Kombinationen der angegebenen Möglichkeiten liefern Werte, die sich aus der Addition der einzelnen <Tasten>-Drücke ergeben (z.B. 67 = 1 + 2 + 64 = *'CapsLock'* ist aktiv und gleichzeitig werden beide <Shift>-Tasten gedrückt gehalten).

Außer dieser Verzweigung hat ON MENU KEY GOSUB keine Funktion. Der Befehlsteil ***GOSUB*** kann weggelassen werden, er wird vom Interpreter automatisch ergänzt. Wird ein nicht existierender ***'Prozedurname'*** angegeben, kann dadurch erreicht werden, daß die z.Zt. aktive Tastatur-Überwachung abgeschaltet wird.

Beachten Sie dazu bitte das ausführliche 'BEISPIEL - PROGRAMM' im ANHANG.

ON MENU MESSAGE GOSUB — Proc.-Bestimmung (Fenster-Event)

```
ON MENU MESSAGE GOSUB Prozedurname
```

Der **ON MENU**-Befehl überwacht die möglichen Fenster-Ereignisse und verzweigt gfls. zu der durch ON MENU MESSAGE

GOSUB ***'Prozedurname'*** angegebenen **PROCEDURE.** Über den **MENU()**-*Ereignispuffer* kann dann dem jeweiligen Fenster-Ereignis entsprechend reagiert werden (s. Ereignistabelle bei **MENU()**).

Außer dieser Verzweigung hat ON MENU MESSAGE GOSUB keine Funktion. Der Befehlsteil ***GOSUB*** kann weggelassen werden, er wird vom Interpreter automatisch ergänzt. Wird ein nicht existierender ***'Prozedurname'*** angegeben, kann dadurch erreicht werden, daß die z.Zt. aktive Fenster-Überwachung abgeschaltet wird.

Die Reaktionsmöglichkeiten und Verfahrensweisen bei der Verwaltung der vielfältigen Fenster-Ereignisse sind im 'BEISPIEL-PROGRAMM' im ANHANG ausführlich beschrieben.

16.5. MENÜ - PROGRAMMIERUNG

MENU Menütext$() — Pulldown-Menü erstellen

```
MENU Menütext$()
```

Dieser Befehl erstellt im Grafik-Modus ein Pulldown-Menu und aktiviert es anschließend.

'Feld$()' ist dabei ein eindimensionales Textfeld, in welchem die Menütitel und Menüpunkt-Einträge der gesamten Menüleiste aufeinanderfolgend eingetragen werden. Zwischen den einzelnen Menüs gelten Leerstrings als Trennmarkierung. Der Aufbau eines solchen Feldes sieht folgendermaßen aus (bei **OPTION BASE 0**):

```
Feld$(0)    =   Titel zu Menü 1  (links beginnend)
Feld$(1)    =   erster Menüpunkt-Eintrag zu Menü 1
Feld$(2)    =   zweiter Menüpunkt-Eintrag zu Menü 1
..............  beliebige Eintragsanzahl für Menü 1
Feld$(n)    =   " "      <= Schlußmarke für Menü 1
.
Feld$(n+1) =    Titel zu Menü 2 (zweites von links)
Feld$(n+2) =    erster  Menüpunkt-Eintrag zu Menü 2
Feld$(n+3) =    zweiter Menüpunkt-Eintrag zu Menü 2
..............  beliebige Eintragsanzahl für Menü 2
Feld$(n+n) =    " "      <= Schlußmarke für Menü 2
.
..............  beliebige Menü-Anzahl für Menüleiste
.
Feld$(nn+1)=    Titel zum letzten Menü (rechts endend)
```

Feld$(nn+2)= erster Menüpunkt-Eintrag zu Menü 'nn'
Feld$(nn+3)= zweiter Menüpunkt-Eintrag zu Menü 'nn'
.... beliebige Eintragsanzahl für Menü 'nn'
Feld$(nn+n)= " " <= Schlußmarke für Menü 'nn'

.
Feld$(nn+n+1)= " " <=Schlußmarke für gesamtes Menü

Den Abschluß für das gesamte Text-***'Feld$()'*** bildet - wie zwischen den einzelnen Menüs - ein weiterer Leerstring (hier: `Feld$(nn+n+1)=""`).

z.B.:

```
DIM Menutxt$(16)           // DIM Menütextfeld
ON MENU GOSUB Auswertung   // MENU-Event abfangen
REPEAT                        // Lese-Schleife
  READ Menutxt$(I%)           // Eintrag lesen
  EXIT IF Menutxt$(I%)="XXX"//EXIT bei DATA-Ende
  I%++                        // Index-Increment
LOOP
Menutxt$(I%)=""               // Menü-Endmarke
'
DATA MENÜ1,  Punkt_1,  Punkt_2,  Punkt_3,""
DATA MENÜ2,  Punkt_4,  Punkt_5,  Punkt_6,""
DATA MENÜ3,  Punkt_7,  Punkt_8,  Punkt_9,""
DATA XXX
'
MENU Menutxt$()               // Menü installieren
DO                            // Main-Loop
  ON MENU                     // Event-Abfrage
LOOP
'
PROCEDURE Auswertung          // Auswertungsroutine
  IF MENU(1)=20               // Menü-Ereignis?
    IF Menutxt$(MENU(0))=" Punkt_1"
      .
      ... Bearbeitung, falls ' Punkt_1' gewählt
      .
    ELSE IF Menutxt$(MENU(0))=" Punkt_2"
      .
      ... Bearbeitung, falls ' Punkt_2' gewählt
      .
    ELSE IF Menutxt$(MENU(0))=" Punkt_x"
      .
      ... weitere Menü-Auswertung
      .
    ENDIF
  ENDIF
RETURN
```

Zur Feststellung, ob ein Menü-Ereignis stattgefunden hat, kann - wie hier in Zeile 1 der Auswertungsroutine - der Eventpuffer-Eintrag **MENU(1)** abgefragt werden. Ein Menü-Ereignis hinterläßt in **MENU(1)** den Wert 20. Wird bei **ON MENU GOSUB** eine spezielle Auswertungsroutine nur für die Behandlung von Menü-

Ereignissen angegeben, so ist diese Abfrage in der Auswertungsprozedur nicht nötig, da diese spezielle Routine dann n u r angesprungen wird, wenn tatsächlich ein *MENU-Event* aufgetreten ist.

Nach einem **'MENU(1)=20'**-Ereignis wird in **MENU(0)** ein Wert geliefert, der dem Feld-Index des gewählten Menüpunktes im Menütextfeld (hier: `'Menütxt$()`) entspricht. Zur gleichen Zeit liegen in **MENU(7)** der Index des geöffneten Pulldown-Menüs (***1*** bis ***'n'***) und in **MENU(8)** der Index des gewählten Menüpunktes dieses Menüs (***1*** bis ***'n'***). Ein Wert von 3 in **MENU(7)** und ein Wert von 6 in **MENU(8)** würde also bedeuten, daß aus dem dritten Pulldown-Menü (von links) der sechste Menüpunkt (von oben) gewählt wurde.

Ebenso wie bei **ALERT** und **POPUP** kann auch in einem MENU zu jedem einzelnen Eintrag ein Tastatur-Kürzel (*'Hotkey'*) bestimmt werden, indem vor dem entsprechenden Zeichen im Menüpunkt-Eintrag ein Tiefstrich plaziert wird. Wird dagegen einem Menüpunkt-Eintrag ein Bindestrich vorangestellt, wird dieser im entsprechenden Pulldown-Menü als nicht aktiv (helle Schrift) dargestellt. Er ist zwar immer noch wählbar, aber seine Auswahl kann dann z.B. als *'nicht getätigt'* ignoriert werden.

Steht keine Maus zur Verfügung, kann das Menü jederzeit durch die Funktionstaste <F1> aufgerufen und anschließend der gewünschte Menüpunkt durch die vier Scroll-<Pfeiltasten> aktiviert werden. Ist der Menüpunkt aktiviert (invertiert), kann er durch <Return> ebenso gewählt werden, als wenn er mit der Maus angeklickt worden wäre.

Noch drei Tips:

- In der Beispiel-Prozedur 'Auswertung' wird nach den Texten der einzelnen Menüpunkte gefragt (`IF Menutxt$(MENU(0)="Menüpunkt-Text")`. Hierbei muß zwar streng darauf geachtet werden, daß in der Abfrage auch dieselben (!) Texte wie in den Menütxt$()-Einträgen verwendet werden, aber diese Lösung ist zu empfehlen, da so sichergestellt wird, daß auch dann noch korrekt verzweigt wird, wenn sich die Reihenfolge oder der Index der Text-Einträge ändern sollte.

- Im obigen Beispiel wurde zudem vor den jeweiligen Menüpunkt-Einträgen zwei Leerzeichen Platz gelassen. Der Sinn liegt darin, daß man - falls Bedarf besteht - vor einem Menüpunkt, der schon aktiviert ist, ein 'Checkmark' (z.B. `MID$(Menütxt$(3),1,1)=CHR$(16)`) setzen kann, ohne daß dadurch die 'Flucht' der Einträge verunstaltet wird.

- Außerdem ist es möglich, den Text eines Menüpunktes während des Programmlaufs zu ändern. Heißt z.B. ein Menüpunkt zur Einstellung von Textattributen 'Fettschrift' und dieser Punkt wird angewählt, um das Textattribut 'fett' zu aktivieren, kann man daraufhin den Eintragstext im entsprechenden Element des Menütextfeldes z.B. auf 'Normalschrift' ändern (z.B. vorher: `Feld$(x)="Fettschrift"`*, dann:* `Feld$(x)="Normalschrift"`*) und das geänderte Menü erneut durch den Befehl 'MENU Feld$()' installieren.*

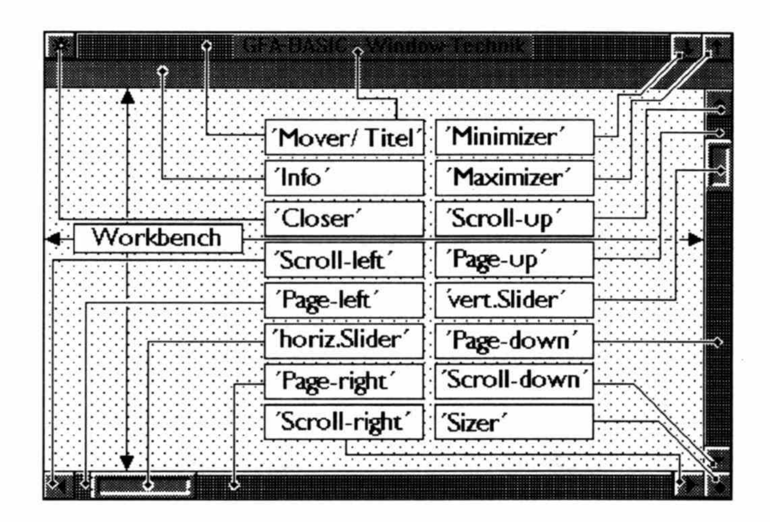
MENU KILL Pulldown-Menü deaktivieren

```
MENU KILL
```

Nach MENU KILL ist das Pulldown-Menü deaktiviert und die aktuelle **ON MENU GOSUB...**-Anweisung wird nicht mehr berücksichtigt. Der Menüleisten-Text wird allerdings dabei nicht vom Bildschirm gelöscht.

16.6. FENSTER - PROGRAMMIERUNG

Wichtig:
In diesem Kapitel finden Sie keine Verweise auf Demo-Listings. Beachten Sie deshalb bitte zu den Fenster-, MENU- und ON MENU-Befehlen das ausführliche 'BEISPIEL - PROGRAMM' im ANHANG.



In deutsch:

´Mover/Titel´	=	Bewegungs- und Titelfeld
´Info´	=	Informationszeile
´Closer´	=	Schließfeld
´Scroll-left´	=	Links-Rollfeld
´Page-left´	=	Links-Seiten-Blätterfeld
´horiz. Slider´	=	horizontaler Schiebebalken
´Page-right´	=	Rechts-Seiten-Blätterfeld
´Minimizer´	=	Fenster-Minimierungsfeld
´Maximizer´	=	Fenster-Maximierungsfeld
´Scroll-up´	=	Aufwärts-Rollfeld
´Page-up´	=	Aufwärts-Seiten-Blätterfeld
´vert. Slider´	=	vertikaler Schiebebalken
´Page-down´	=	Abwärts-Seiten-Blätterfeld
´Scroll-down´	=	Abwärts-Rollfeld
´Sizer´	=	Größen-Veränderungsfeld
´Workbench´	=	Fenster-Arbeitsbereich

CLEARW # — Fenster-Inhalt löschen

```
CLEARW [#]Nummer
```

'Nummer' bestimmt im Grafikmodus die Nummer des GFA-Fensters (0 bis 4), dessen Inhalt gelöscht werden soll. Ist kein Fenster mit der angegebenen ***'Nummer'*** geöffnet, bleibt der Befehl ohne Wirkung.

Im aktuellen Fenster ist ein Löschen des Inhalts auch durch **CLS** möglich.

CLIP # { CLI } — Ausgabe auf Fensterbereich begrenzen

```
CLIP #Nummer
```

Bewirkt eine Beschränkung von Grafikausgaben auf den Arbeitsbereich des durch **OPENW#** geöffneten GFA-Fensters mit der angegebenen ***'Nummer'*** (0 bis 4). Die Position und Ausmaße der Arbeitsfläche dieses Fensters werden dann als *Clip*-Rechteck installiert, sodaß Grafikausgaben außerhalb dieses Rechtecks unterdrückt werden. **CLIP #0** wirkt wie **CLIP OFF**. Sollen Grafikausgaben auf das Hintergrundfenster #0 beschränkt werden, ohne daß die Menüleiste überschrieben werden kann, muß gfls. erst **CLIP OFF** und dann

```
CLIP 0,Leistenhöhe,Screenbreite,Screenhöhe-Leistenhöhe
```

eingesetzt werden.

CLOSEW # { CL W } Fenster schließen

```
CLOSEW [#]Nummer
```

'Nummer' bestimmt im Grafikmodus die Nummer des zu schließenden GFA-Fensters (0 bis 4). Ist kein Fenster mit der angegebenen ***'Nummer'*** geöffnet, bleibt der Befehl ohne Wirkung.

FULLW { FUL } Fenster auf Bildschirmgröße maximieren

```
FULLW [#]Nummer
```

'Nummer' bestimmt die Nummer des GFA-Fensters (1 - 4), welches bis an die - gfls. vorhandene - Menüzeilen-Aussparung am oberen Rand vergrößert werden soll. Ist kein Fenster mit der angegebenen ***'Nummer'*** geöffnet, bleibt der Befehl ohne Wirkung.

GETFIRST # { GETF } Rechteckliste initialisieren

```
GETFIRST [#]Nummer,Xp&,Yp&,Br&,Ho&
```

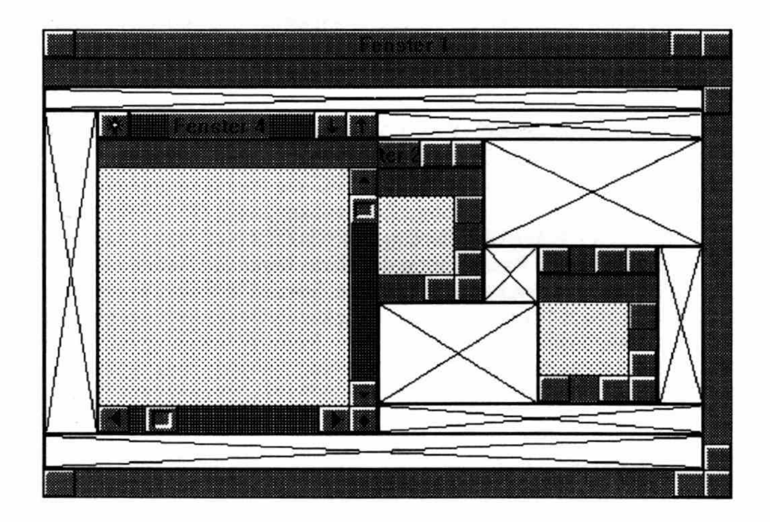
Bei den meisten Fensteroperationen (Löschen, Verschieben, Größenänderung etc.) ergibt es sich, daß verschiedene Rechteck-Bereiche innerhalb von Fenstern oder im umgebenden Bildschirm neu gezeichnet werden müssen. Für diesen Vorgang wird im allgemeinen der englische Ausdruck *'Redraw'* (Neuzeichnung) verwendet. In GFA-BASIC wird bei den entsprechenden Fensteroperationen - falls **ON MENU, GETEVENT** oder **PEEKEVENT** eingesetzt wurde - in **MENU(1)** ein *'Redraw'*-Ereignis mit der Kennziffer 21 gemeldet (s. **MENU()**-*Ereignispuffer*).

Die Feststellung, welche Bereiche von diesem Ereignis betroffen sind, ist relativ kompliziert und würde *'per Hand'* (mittels **RC_INTERSECT()**) doch einen erheblichen Programmieraufwand bedeuten. GFA-BASIC bietet daher den GETFIRST-Befehl an, der für alle durch **OPENW** geöffneten Fenster eine sog. *'Rechteckliste'* generiert. In dieser Rechteckliste sind alle sichtbaren - also nicht durch andere Fenster verdeckten - Rechteck-Bereiche des betreffenden Fensters durch die Koordinaten der linken oberen Ecke, sowie durch ihre Höhe und Breite beschrieben.

Ist die Rechteckliste für das durch den Parameter ***'Nummer'*** angegebene GFA-Fenster generiert, liefert GETFIRST in den zu übergebenden Integer-Rückgabevariablen ***'Xp&'*** und ***'Yp&'*** die

absoluten - auf die linke, obere Bildschirmecke bezogenen - Koordinaten der linken oberen Ecke, sowie in ***'Br&'*** und ***'Ho&'*** die Breite und Höhe des ersten Rechtecks dieser Liste. Ist das Fenster vollständig durch andere Fenster verdeckt, so steht in ***'Br&'*** und ***'Ho&'*** eine Null. Es besteht also keine Notwendigkeit, dieses Fenster dann neu zu zeichnen.

Die Rechteckliste von Fenster 1:



Bei Fenstern, die nur teilweise von anderen Fenstern verdeckt werden, ergeben sich logischerweise mehrere sichtbare Rechtekke, die neu gezeichnet werden müssen. In diesen Fällen müssen durch den Befehl **GETNEXT** (s. dort) die Koordinaten und Maße der weiteren Redraw-Rechtecke ermittelt werden.

GETNEXT # { GETN } nächstes Listen-Rechteck ermitteln

```
GETNEXT Xp&,Yp&,Br&,Ho&
```

Wie unter **GETFIRST** beschrieben, kommt es bei verschiedenen Fenster-Ereignissen zu der Notwendigkeit, Rechteck-Bereiche von Fenstern neu zeichnen zu müssen. **GETFIRST** erstellt zu dem jeweiligen Fenster eine Liste aller davon betroffenen Rechtecke und liefert die Koordinaten und Maße des ersten Rechtecks dieser Liste. Ergeben sich aus den Überschneidungen der Fenster mehrere Redraw-Bereiche des betroffenen Fensters, so können die weiteren Rechtecke mit GETNEXT ermittelt werden.



Dazu werden - wie auch bei **GETFIRST** - vier Integer-Rückgabevariablen übergeben, die anschließend die absoluten Koordinaten der linken, oberen Ecke des Rechtecks (***'Xp&'*** und ***'Yp&'***), sowie dessen Breite und Höhe (***'Br&'*** und ***'Ho&'***) enthalten. Diese Abfrage wird so oft wiederholt, bis in ***'Br&'*** und ***'Ho&'*** eine Null zurückgegeben wird. Ist dies der Fall, so war das vorhergehend ermittelte Rechteck das letzte in der jeweiligen Liste. Wird bereits die erste GETNEXT-Abfrage in ***'Br&'*** und ***'Ho&'*** mit Null beantwortet, so war das durch **GETFIRST** ermittelte Rechteck die Vollfläche entweder des aktiven Fensters oder eines inaktiven und frei liegenden - also nicht verdeckten - Fensters.

Wurde durch **OPENW #0** ein Hintergrund-Fenster geöffnet und durch **SYSCOL** (*'Objekt' 6 und 7*) Farben und Füllmuster dafür bestimmt, so restauriert GFA-BASIC die im Hintergrundfenster liegenden Rechtecke nötigenfalls selbstständig. **GETFIRST** und GETNEXT lassen sich allerdings auch auf das Null-Fenster anwenden, so daß die Restauration der darin liegenden Redraw-Rechtecke selbst verwaltet werden kann. Speziell in Fällen, in welchen auf dem Null-Fenster grafische Objekte als Desktop-Icons plaziert werden, empfiehlt es sich, den Redraw-Vorgang auf dem selbstdefinierten Desktop zu überwachen und Icons, sowie gfls. ähnliche Objekte neu zu zeichnen.

INFOW # { INF } Fenster-Informationszeile bestimmen

```
INFOW [#]Nummer,''Text''
```

Einem GFA-Fenster kann eine Informationszeile zugeordnet werden, die sich dann unterhalb der Titelzeile (*'Mover'*) befindet. ***'Nummer'*** bestimmt die Nummer des betreffenden Fensters (1 bis 4). Die weiteren Erläuterungen unter **TITLEW** gelten sinngemäß auch für INFOW.

MOVEW # { MOV } Fenster bewegen

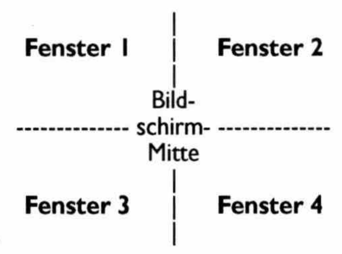
```
MOVEW [#]Nummer,Xp,Yp
```

MOVEW zeichnet das - vorher durch **OPENW** geöffnete - GFA-Fenster mit der angegebenen Fenster-***'Nummer'*** (1 bis 4) neu, so daß die linke, obere Ecke des äußeren Fensterumrisses auf den absoluten Koordinaten ***'Xp'/'Yp'*** zu liegen kommt. Es werden dabei die jeweils aktiven Randelemente des Fensters neu gezeichnet. GFA-BASIC übernimmt dabei auch die Verlegung des aktuellen Fenster-Inhaltes.

OPENW #{o w} Fenster öffnen

```
OPENW [#]Nummer [,Xp,Yp,Br,Ho,Attribute]
```

'Nummer' bestimmt das zu öffnende, bzw. zu aktivierende Fenster. Wird nur `OPENW #'Nummer'` ohne Angabe der Koordinaten und Attribute verwendet, wird das angegebene Fenster mit allen Attributen nach folgendem Schema positioniert:



Wenn diese Bildschirm-Aufteilung nicht gewünscht wird, können mit den optionalen Parametern ***'Xp'/'Yp'*** die linke, obere Ecke sowie durch ***'Br'*** und ***'Ho'*** die Breite und Höhe des Fensterumrisses bestimmt werden.

In diesem Fall wird durch den Parameter ***'Attribute'*** festgelegt, mit welchen Randelementen das Fenster ausgestattet werden soll. ***'Attribute'*** stellt einen Bit-Vektor dar, in welchem durch Setzen der jeweiligen Bits die dazugehörigen Elemente angeschaltet werden:

-1 (alle Bits an)	=	alle Attribute zeichnen
0 (alle Bits aus)	=	keine Attribute (es wird ein 3D-Rechteck gezeichnet)

Bit0:	1	=	der vertikale Schieberegler
Bit1:	2	=	die vertikalen Scrollpfeile
Bit2:	4	=	der horizontale Schieberegler
Bit3:	8	=	die horizontalen Scrollpfeile
Bit4:	16	=	*'Mover'*-Balken incl. Titelzeile
Bit5:	32	=	*'Closer'*-Button links oben
Bit6:	64	=	*'Minimizer'*-Button rechts oben
Bit7:	128	=	*'Maximizer'*-Button rechts oben
Bit8:	256	=	*'Info'*-Zeile unter dem *'Mover'*
Bit9:	512	=	*'Sizer'*-Button rechts unten

Die Reaktionsmöglichkeiten und Verfahrensweisen bei der Verwaltung der einzelnen Fenster-Ereignisse sind im 'BEISPIEL-PROGRAMM' im Anhang ausführlich beschrieben.

Um den Hintergrund der Fenster 1 bis 4 einfacher verwalten zu können, sollte grundsätzlich vor dem ersten Öffnen eines dieser Fenster das - nicht sichtbare - Fenster mit der Nummer 0 geöffnet werden. Dieses Fenster hat die Aufgabe, den Menüzeilenbereich am oberen Bildrand auszusparen und den Koordinatenursprung um die Menüzeilenhöhe nach unten zu versetzen. Grafik- und/ oder **PRINT**-Ausgaben werden dann am unteren Rand der Menüzeile ge'*clipt*', sodaß ein Überschreiben des Menüs verhindert wird.Außerdem bietet **SYSCOL** für das Null-Fenster zusätzlich eine vereinfachte Möglichkeit zur Restauration des Bildschirm-Hintergrundes.

SIZEW # { SIZ } neue Fenster-Größe bestimmen

```
SIZEW [#]Nummer,Br,Ho
```

SIZEW zeichnet die aktiven Randelemente des - vorher durch **OPENW** geöffneten - GFA-Fensters mit der angegebenen ***'Nummer'*** (1 bis 4) neu, sodaß es anschließend in den äußeren Umrissen die in ***'Br'*** angegebene Breite und die in ***'Ho'*** angegebene Höhe aufweist . Die linke, obere Ecke des Fensterumrisses bleibt dabei auf der alten Position liegen. Bei Fenster-Vergrößerung wird gfls. ein Redraw-Ereignis für das betroffene Fenster ausgelöst (**MENU(1)=21**). Der alte Fensterinhalt bleibt an der bisherigen Position erhalten. Die Arbeitsfläche in den neu hinzukommenden Rechtecken muß anschließend restauriert werden.

Bei Fenster-Verkleinerung wird für die gfls. darunter liegenden anderen Fenster ein Redraw-Ereignis in **MENU(1)** gemeldet. Der Fenster-Inhalt des in der Größe veränderten Fensters bleibt innerhalb der neuen Arbeitsfläche erhalten.

TITLEW # { TIT } Fenster-Titelzeile bestimmen

```
TITLEW [#]Nummer,''Text''
```

Das Fenster mit der angegebenen ***'Nummer'*** erhält im Titelbalken ('*Mover*') den angegebenen ***'Text'*** als Überschrift.

Die Titelzeile kann auch nachträglich eingesetzt werden, bzw. während des Programmlaufs beliebig geändert werden, ohne dazu das Fenster schließen und wieder öffnen zu müssen. Es muß allerdings beim Öffnen des betreffenden Fensters in der 'Attribut'-Bestimmung (s. **OPENW #**) eine Titelzeile initialisiert worden sein. Der geänderte Titel-Text wird im aktiven Fenster sofort und bei inaktiven Fenstern mit dem nächsten Aktivieren des Fensters (z.B. durch **TOPW**) sichtbar.

TOPW # { TO } Fenster-Parameter setzen und aktivieren

```
TOPW [#]Nummer
```

Dieser Befehl aktiviert das GFA-Fenster mit der angegebenen ***'Nummer'*** (1 bis 4). Das Fenster muß dazu vorher durch **OPENW** geöffnet worden sein. Liegt es vor TOPW ganz oder teilweise unter einem anderen Fenster, wird ein Redraw-Ereignis in **MENU(1)** gemeldet, das Fenster *'nach oben'* gelegt und dann aktiviert. Die Randelemente werden als wählbar gezeichnet, der Koordinaten-Nullpunkt in die linke, obere Ecke der Arbeitsfläche gelegt und das Grafik-*'Clipping'* für das Fensters angeschaltet.

Außerdem werden in den reservierten GFA-Variablen **'_X'** und **'_Y'** die Breite und Höhe der aktuellen Arbeitsfläche und in der **WINDGET**-Tabelle die für dieses Fenster spezifischen Parameter eingetragen Auch Datenausgaben mittels **PRINT** oder **CRSCOL-/CRSLIN**-Abfragen etc. beziehen sich nun auf die neue Arbeitsfläche.

WIN # { WI } Fenster-Parameter setzen

```
WIN [#]Nummer
```

Dieser Befehl ist prinzipiell identisch mit **TOPW** (s. dort). Der einzige Unterschied ist der, daß das mit ***'Nummer'*** angegebene Fenster nicht aktiviert wird. Die Randelemente des Fensters werden also nicht neu gezeichnet und das vorher aktive Fenster bleibt für den Anwender als aktiv sichtbar.

Dieser Befehl ist notwendig, um bei einem Redraw-Vorgang dieses Fensters so tun zu können, als wäre es das aktive Fenster. Die reservierten Variablen **'_X'** und **'_Y'**, sowie die Einträge in der **WINDGET**-Tabelle werden aktualisiert und können nun für die Neuzeichnung des Fensters gelesen und verwendet werden.

WINDFIND { WINDF } Fensternummer ermitteln

```
WINDFIND Xp,Yp,Var&
```

Liefert in der Integer-Rückgabevariablen ***'Var&'*** die GFA-Fensternummer (1 bis 4) des Fensters, das sich unter der Bildschirmposition mit den absoluten Koordinaten ***'Xp'/'Yp'*** befindet. Wird eine Null geliefert, so befindet sich kein Fenster an der

angegebenen Position. Werden dagegen an der angegebenen Position mehrere Fenster untereinander gefunden, so wird in ***'Var&'*** die Nummer des zuoberst liegenden Fensters geliefert.

WINDGET { WINDG } Fenster-Parameter gesamt lesen

```
WINDGET Index,Var1& [,Var2&,Var3&,...]
```

Der Befehl WINDGET ermöglicht das Lesen der fensterspezifischen Parameter-Tabelle des aktuellen Fensters (s. **TOPW** und **WIN**). Dazu wird in ***'Index'*** ein Startelement angegeben. Beginnend mit diesem Element werden dann den hinter ***'Index'*** aufgeführten Integer-Rückgabevariablen (***'Var1&'***,***'Var2&'***,... etc.) der Reihe nach soviele Werte aus der Tabelle zugewiesen, wie Variablen in der Liste vorhanden sind.

z.B.: `WINDGET 8,Slidepos&,Slidesize&`

überträgt den Inhalt des achten Elements in die Variable ***'Slidepos&'*** und den Inhalt des neunten Elements in die Variable ***'Slidesize&'***.

'Index':

	0	=	absolute X-Koordinate der linken, oberen Ecke des Gesamtfensters
	1	=	absolute Y-Koordinate der linken, oberen Ecke des Gesamtfensters
	2	=	Breite des Gesamtfensters
	3	=	Höhe des Gesamtfensters
	4	=	absolute X-Koordinate der linken, oberen Ecke der Fenster-Arbeitsfläche
	5	=	absolute Y-Koordinate der linken, oberen Ecke der Fenster-Arbeitsfläche
	6	=	Breite der Fenster-Arbeitsfläche
	7	=	Höhe der Fenster-Arbeitsfläche
=>	8	=	Stellung des vertikalen Schiebereglers: (0 = ganz oben ; 1000 = ganz unten)
=>	9	=	Größe des vertikalen Schiebereglers (0-1000)
=>	10	=	Stellung des horizontalen Schiebereglers: (0 = ganz links ; 1000 = ganz rechts)
=>	11	=	Größe des horiz. Schiebereglers (0-1000)
	12	=	aktive Fenster-Attribute (Bit-Vektor s. **OPENW**)
=>	13	=	**WINDSET**-Attribute des aktuellen Buttons
=>	14	=	aktuelle Texthöhe (8, 14 oder 16)
=>	15	=	aktuelle Zeichensatz-Adresse (Segment:Offset -> Longword, s. **LOADFONT**)
	16	=	GFA-Fensternummer des obersten Fensters
	17	=	GFA-Fensternummer des zweitobersten Fensters
	18	=	GFA-Fensternummer des zweituntersten Fensters
	19	=	GFA-Fensternummer des untersten Fensters

Die mit einem Pfeil gekennzeichneten Einträge können durch **WINDSET** auch gesetzt werden (s. dort).

WIND_GET() Fenster-Parameter separat lesen

```
Var=WIND_GET(Index)
```

WIND_GET() als Funktion liefert separat den Wert, der in der **WINDGET**-Tabelle des aktuellen Fensters (s. **TOPW** und **WIN**) unter dem angegebenen ***'Index'*** enthalten ist. Weitere Erläuterungen finden Sie bei dem Befehl **WINDGET**.

WINDSET { WINDS } Fenster-Parameter ändern

```
WINDSET Index,Wert1 [,Wert2,Wert3,...]
```

Durch WINDSET können Änderung an einigen Elementen der **WINDGET**-Tabelle vorgenommen werden. Dazu wird in ***'Index'*** ein Startelement angegeben. Beginnend mit diesem Element werden dann die hinter 'Index' aufgeführten Werte (***'Wert1','Wert2',...*** etc.) der Reihe nach sovielen Tabelle-Elementen zugewiesen, wie Werte in der Liste vorhanden sind.

z.B.: `WINDSET 8,660,120`

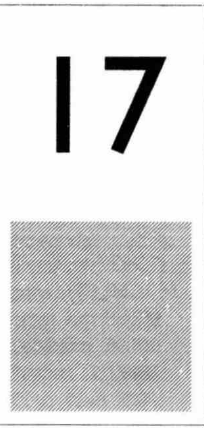
überträgt den Wert 660 in das achte Element der Tabelle (Position des vertikalen Schiebereglers) und den Wert 120 in das neunte Element der Tabelle (Größe des vertikalen Schiebereglers).

'Index':

8	=	Stellung des vertikalen Schiebereglers: (0 = ganz oben ; 1000 = ganz unten)
9	=	Größe des vertikalen Schiebereglers (0-1000)
10	=	Stellung des horizontalen Schiebereglers: (0 = ganz links ; 1000 = ganz rechts)
11	=	Größe des horizontalen Schiebereglers (0-1000)
13	=	Attribute des aktuellen Fenster-Buttons setzen
14	=	Texthöhe setzen (nur *EGA/VGA*: 8, 14 oder 16) Hierzu bitte auch **DEFTEXT** und **LOADFONT** beachten.
15	=	neue Zeichensatz-Adresse setzen (Segment: Offset => Longword, s. **LOADFONT**) Hierzu bitte auch **DEFTEXT** und **LOADFONT** beachten.

Notizen:

Anhang



ANHANG A. 'BEISPIEL-PROGRAMM'

WINDOW-Demo

```
SCREEN 18                      // Für VGA
'
CLS                            // Klar Schiff
scr_x%=_X , scr_y%=_Y          // Bildschirm-Auflösung
indx%=4                        // 4 Fenster
w_par%=$FFFFFFFF               // Bitvector für Fensterstatus
DIM x%(4),y%(4),b%(4),h%(4)    // 4 Felder für Fenster-
'                              // Koordinaten und Maße
m_set                          // Pulldown-Menü-Init
ON MENU GOSUB m_deal           // Event-Kontrolle
ON MENU BUTTON GOSUB m_essage  // für Menü, Maus,
ON MENU MESSAGE GOSUB m_essage // Fenster und
ON MENU KEY GOSUB k_ey         // Tastatur einschalten
SYSCOL 6,1,7                   // Farbe und Füllmuster für
SYSCOL 7,4,4                   // Bildschirm-Hintergrund
OPENW #0                       // Sesam-öffne-Dich-Screenfenster
OPENW #1                       // Einmal ein Window...
FULLW #1                       // ...voll öffnen und...
CLOSEW #1                      // ...gleich wieder schließen:
'                              // Bildschirm-Hintergrund okay!
ALERT 2,''CPU:'',3,''8088|286|386sx|386|486",intel%
'                              // Bremsfaktor
FOR I%=4 DOWNTO 1              // vier Fenster
  INFOW #I%,'' INFO-Zeile ''+STR$(I%)      // Infozeile setzen
  TITLEW #I%,'' TITEL-Zeile ''+STR$(I%)    // Titelzeile setzen
  attr%=1008                   // keine Scrollelemente
  IF I%=1 THEN attr%=TRUE      // außer in window 1
  OPENW #I%,100-I%*20+30,80-I%*16+20,scr_x%/2,scr_y%/2,attr%
  w_cls(I%)                    // 'n bißchen Farbe rein
NEXT I%                        // nächstes Fenster
DO                             // Hauptschleife
  ON MENU                      // Event-Kontrolleur
  '                            // witzige bouncing-box
  IF (TIMER-cnt%)>1000         // genug Zeit für Fensterhandling
    IF WIND_GET(16)=1          // Fenster 1 auf?
      MOUSE mx%,my%,k%         // Maus-Koord. holen
      COLOR indx%*3            // Fensterfarbe
      DEFFILL indx%*7          // Fenstermuster
      PBOX px1%,py1%,px1%+30,py1%+30       // Quellbox zeichnen
```




```
      px1%=px2%, py1%=py2%        // Koordinaten-Eimerkette
      px2%=RAND(_X-30), py2%=RAND(_Y-30)  // neue Ziel-Koord.
      gsbox(7,10,5,px1%,py1%,30,30,px2%,py2%,30,30,15,15)
      '                           // und los geht's
     DEFFILL RAND(24)+10          // Muster und...
     COLOR RAND(15)               // ...Farbe für neu Zielbox
     PBOX px2%,py2%,px2%+30,py2%+30 // Zielbox zeichnen
     IF mx%<>MOUSEX OR my%<>MOUSEY OR MOUSEK THEN cnt%=TIMER
      '                           // Maus-Aktion ?? Dann Pause
    ENDIF
  ENDIF
LOOP                              // Hauptschleifen-Ende
PROCEDURE m_essage                // Window-Message-Handling
  WINDGET 16,indx%                // Index des akt. Windows holen
   wg8%=WIND_GET(8),wg9%=WIND_GET(9)      // Slider-Größe und
   wg10%=WIND_GET(10),wg11%=WIND_GET(11)  // -Position holen
  SELECT MENU(1)                  // Ereignis-Typ feststellen
  CASE 2                          // TOP-Window
    IF MENU(7)                    // Window gewählt?
      indx%=MENU(7)               // Index feststellen
      INFOW #indx%,''(TOPPER) Window ''+STR$(indx%)+'' aktiviert''
      IF BTST(w_par%,indx%-1)=0       // Window ist 'minimized'
        w_par%=BSET(w_par%,indx%-1)  // Aktiv-Flag anschalten
        CLOSEW #indx%             // erst schließen
        attr%=1008                // dann Attribute...
        IF indx%=1 THEN attr%=TRUE  // ...setzen
        OPENW #indx%,x%(indx%),y%(indx%),b%(indx%),h%(indx%),attr%
        '                         // neu öffnen
      ENDIF
      TOPW #indx%                 // 'toppen'
      w_cls(indx%)                // und Tapete drauf
      gsbox(5,6,10,0,0,_X,_Y,0,0,_X,_Y,0,0)       // 'grow'-Effekt
    ENDIF
  CASE 4                          // CLOSE-Window
    gsbox(1,6,50,0,0,_X,_Y,0,0,0,0,0,0)           // 'shrink'-Effekt
    w_par%=BCLR(w_par%,indx%+3) // 'closed'-Flag setzen
    WINDGET 0,x%(indx%),y%(indx%),b%(indx%),h%(indx%)
    '                             // die alten Koordinaten merken
    CLOSEW #indx%                 // und Fenster zu
    IF WIND_GET(16)               // noch ein Fenster da?
      TOPW #WIND_GET(16)          // dann bitte Bewegung
    ELSE                          // alle Fenster zu!
      ALERT 2,''Programm-Ende ?'',1,''OKAY|NEIN'',d%
      IF d%=1                     // Programm-Ende?
        EDIT                      // zurück zum Editor
      ELSE                        // weiter !
```

```
      ALERT 1,''Fenster öffnen:'',5,''1|2|3|4|1-4",indx%
    ENDIF
    IF indx%=5                 // alle Fenster öffnen
      FOR indx%= 1 TO 4        // vier Stück
        OPENW #indx%           // aufmachen
        w_cls(indx%)           // und anmalen
      NEXT indx%               // nächstes
    ELSE                       // nur eins öffnen!
      ? ''Maustaste drücken, halten und Maus ziehen''
      REPEAT                   // auf...
      UNTIL MOUSEK             // Maustaste warten
      MOUSE x%,y%,k%           // Maus-Status holen
      RUBBERBOX x%,y%,60,60,b%,h%        // Lasso-Box aufziehen
      attr%=1008               // Attribute für 2 - 4
      IF I%=1 THEN attr%=TRUE // Window 1 mit allen Atrib.
      OPENW #indx%,x%,y%,b%,h%,attr%     // öffnen
      w_cls(indx%)             // und tapezieren
    ENDIF
  ENDIF
CASE 5                         // MINIMIZE-Window
  WINDGET 0,x%(indx%),y%(indx%),b%(indx%),h%(indx%)
  '                            // die alten Koordinaten merken
  CLOSEW #indx%                // schließen
  OPENW #indx%,(indx%)*40,scr_y%-40,30,30,0   // als kleine
  '                            // Box wegpacken ('iconisieren')
  CLEARW #indx%                // und putzen
  w_par%=BCLR(w_par%,indx%-1) // aktiv-Flag ausschalten
  TOPW #WIND_GET(17)           // nächstoberes Fenster 'toppen'
  w_cls(WIND_GET(16))          // und tapezieren
CASE 6                         // MAXIMIZE-Window
  INFOW #indx%,''Volle Gröpe (MAXIMIZER)''
  IF BTST(w_par%,indx%-1)=0   // war Fenster vorher inaktiv
    w_par%=BSET(w_par%,indx%-1)    // dann aktiv-Flag setzen
    CLOSEW #indx%              // Fenster zu
    attr%=1008                 // Attribute setzen
    IF I%=1 THEN attr%=TRUE   // für Fenster 1 alle
    OPENW #indx%,0,0,scr_x%,scr_y%,attr%// und Fenster auf
  ENDIF
  FULLW #indx%                 // auf volle Größe bringen
  gsbox(7,10,10,_X/2-10,_Y/2-10,20,20,0,0,_X,_Y,20,20) // Effekt
  w_cls(indx%)                 // und Farbe rein
CASE 7                         // Aufwärtspfeil angeklickt?
  WINDSET 8,wg8%-20            // vert. Schieber-Pos. vermindern
  INFOW #indx%,''(UP-ARROW) / Pos:''+STR$(wg8%)
CASE 8                         // Abwärtspfeil angeklickt?
  WINDSET 8,wg8%+20            // vert. Schieber-Pos. erhöhen
```




```
    INFOW #indx%,''(DOWN-ARROW) / Pos:''+STR$(wg8%)
  CASE 9                        // Linkspfeil angeklickt?
    WINDSET 10,wg10%-20         // horiz. Schieber-Pos. vermindern
    INFOW #indx%,''(LEFT-ARROW) / Pos:''+STR$(wg10%)
  CASE 10                       // Rechtspfeil angeklickt?
    WINDSET 10,wg10%+20         // horiz. Schieber-Pos. erhöhen
    INFOW #indx%,''(RIGHT-ARROW) / Pos:''+STR$(wg10%)
  CASE 11                       // vert.Balken oben angeklickt?
    WINDSET 9,wg9%-20           // vert. Schiebergröße vermindern
    INFOW #indx%,''(PAGE-UP) / Siz:''+STR$(wg9%)
  CASE 12                       // vert.Balken unten angeklickt?
    WINDSET 9,wg9%+20           // vert. Schiebergröße erhöhen
    INFOW #indx%,''(PAGE-DOWN) / Siz:''+STR$(wg9%)
  CASE 13                       // horiz.Balken links angeklickt?
    WINDSET 11,wg11%-20         // horiz.Schiebergröße vermindern
    INFOW #indx%,''(PAGE-LEFT) / Siz:''+STR$(wg11%)
  CASE 14                       // horiz.Balken rechts angeklickt?
    WINDSET 11,wg11%+20         // horiz.Schiebergröße erhöhen
    INFOW #indx%,''(PAGE-RIGHT) / Siz:''+STR$(wg11%)
  CASE 15                       // vert. Schieber bewegt?
    INFOW #indx%,''(V-SLIDER) / Pos: ''+STR$(MENU(7))
    WINDSET 8,MENU(7)           // neue Position setzen
  CASE 16                       // horiz. Schieber bewegt?
    INFOW #indx%,''(H-SLIDER) / Pos: ''+STR$(MENU(7))
    WINDSET 10,MENU(7)          // neue Position setzen
  CASE 17                       // MOVE-Window
    INFOW #indx%,''Fenster wurde bewegt (MOVER)''
    MOVEW #indx%,MENU(7),MENU(8)     // an die neue Position
    w_cls(indx%)                // Fenster putzen
  CASE 18                       // SIZE-Window
    INFOW #indx%,''Fenstergröpe verändert (SIZER)''
    SIZEW #indx%,MENU(7),MENU(8)     // neue Größe einstellen
    w_cls(indx%)                // Fenster putzen
  CASE 19                       // INFO-Zeile
    INFOW #indx%,''Infozeile wurde angeklickt!''
  CASE 21                       // Redraw-Event ist eingetreten
    @redraw(MENU(7),MENU(8),MENU(9),MENU(10))  // Bereiche...
  ENDSELECT                     //... neu zeichnen
RETURN
PROCEDURE m_deal                // Menü-Handling
  mp%=MENU(0)                   // Menüpunkt-Index?
  IF m_punkt$(mp%)=''  Datei laden''
    FILESELECT ''*.*'',''Meine.Dat'',path$     // Datei auswählen
    @Path(path$,'':'',label$,path$,F$)// Doppelpunkt suchen
    @Path(path$,''\'',path$,b$,F$)      // ersten Backslash suchen
    @Path(b$,''.'',b$,c$,F$)            // nach Trennpunkt suchen
```

```
    al$=''gewählt:|''+label$+'' (Laufwerk)|''+path$+'' (Pfad)|''
    al$=al$+b$+'' (Name)|''+c$+'' (Extension)''
    ALERT 3,al$,0,''Danke'',back%
    ' ** Lade-Routine **
  ELSE IF m_punkt$(mp%)=''  Datei speichern''
    ' ** Speicher-Routine **
  ELSE IF m_punkt$(mp%)=''  Datei einrichten''
    ' ** Init-Routine **
  ELSE IF m_punkt$(mp%)=''  Datei löschen''
    ' ** Lösch-Routine **
  ELSE  IF m_punkt$(mp%)='' Information''
    al$=''Geschlossene Fenster können|über die Ziffern''
    al$=al$+''tasten 1 - 4|wieder geöffnet werden''
    ALERT 1,al$,1,''OKAY'',back%
  ELSE IF m_punkt$(mp%)='' Quit''
    ' ** Quit-Routine **
    EDIT
  ENDIF
  IF m_punkt$(mp%)=''  Menüpunkt 1a''
    ' ** Menü-Routine 1 **
    al$=''Der Text des Menüpunktes 1a|ist jetzt verändert!''
    ALERT 1,al$,1,'' AHA !! '',back%
    m_punkt$(mp%)=''  1a umgekehrt''
    MENU m_punkt$()
  ELSE IF m_punkt$(mp%)=''  1a umgekehrt''
    ' ** Alternative zu Menü-Routine 1 **
    m_punkt$(mp%)=''  Menüpunkt 1a''
    MENU m_punkt$()
  ENDIF
  IF m_punkt$(mp%)=''  Menüpunkt 1b''
    ' ** Menü-Routine 2 **
    al$=''Der Menüpunkt 1b|ist jetzt markiert!''
    ALERT 1,al$,1,'' SoSo !! '',back%
    m_punkt$(mp%)=''• <- 1b-Markierung''
    MENU m_punkt$()
  ELSE IF m_punkt$(mp%)=''• <- 1b-Markierung''
    ' ** Markierung für Menü-Routine 2 aufheben **
    m_punkt$(mp%)=''  Menüpunkt 1b''
    MENU m_punkt$()
  ENDIF
  IF m_punkt$(mp%)=''  Menüpunkt 1c''
    ' ** Menü-Routine 3 **
    al$=''Der Menüpunkt 1c|ist jetzt unbrauchbar !''
    ALERT 1,al$,1,'' Nanu ?? '',back%
```

```
    m_punkt$(mp%)=''- Menüpunkt 1c''
    m_punkt$(mp%+2)=''     1c wieder aktivieren''
    MENU m_punkt$()
    ? mp%
  ENDIF
  IF  m_punkt$(mp%)=''     1c wieder aktivieren''
    m_punkt$(mp%-2)='' Menüpunkt 1c''
    m_punkt$(mp%)=''    Menüpunkt 2a''
    MENU m_punkt$()
  ELSE  IF m_punkt$(mp%)=''     Menüpunkt 2a'' //-- //
    ' ** Menü-Routine 4 **                         //
  ELSE IF m_punkt$(mp%)=''     Menüpunkt 2b''      //
    ' ** Menü-Routine 5 **                         // irgend-
  ELSE IF m_punkt$(mp%)=''     Menüpunkt 2c''      // welche
    ' ** Menü-Routine 6 **                         // Menü-
  ELSE IF m_punkt$(mp%)=''         Menüpunkt 3a''  // punkte
    ' ** Menü-Routine 7 **                         //
  ELSE IF m_punkt$(mp%)=''         Menüpunkt 3b''  //
    ' ** Menü-Routine 8 **                         //
  ELSE IF m_punkt$(mp%)=''         Menüpunkt 3c''  //
    ' ** Menü-Routine 9 **                     //-- //
  ENDIF
RETURN
PROCEDURE k_ey
  IF MENU(1)=1                   // Taste gedrückt%
    taste%=MENU(5) AND $FF       // ASCII-Code holen
    shift%=MENU(6)               // Shift-Status holen
     PRINT ''Shift-/ ASCII-Code : '';shift%'''/'''taste%
    indx%=VAL(CHR$(taste%))      // Tasten-Zeichen ermitteln
    IF indx%=>1 AND indx% <= 4   // Taste '1' bis '4' ?
      IF BTST(w_par%,indx%+3)=0 // betreffendes Fenster inaktiv?
        w_par%=BSET(w_par%,indx%+3)  // Aktiv-Flag setzen
        OPENW #indx%,x%(indx%),y%(indx%),b%(indx%),h%(indx%),attr%
        '                        // Fenster mit alten Koord. öffnen
        w_cls(indx%)             // und anmalen
      ENDIF
    ENDIF
  ENDIF
RETURN
PROCEDURE m_set                  // Pulldown-Menü-Init
  DIM m_punkt$(32)               // hier: max. 32 Einträge
  RESTORE m_datas                // DATA-Zeiger setzen
  FOR i=0 TO 31                  // 32 Einträge
    READ m_punkt$(i)             // lesen
     EXIT IF m_punkt$(i)=''~~~~''    // Exit, wenn Endemarkierung
  NEXT i                         // nächster Eintrag
```

```
  m_punkt$(i)=''''                // Menü-Init abschließen
  m_punkt$(i+1)=''''              //   ''     ''       ''
  m_datas:                        // DATA-Label
  DATA DATEI
  DATA    Datei laden
  DATA    Datei speichern
  DATA    Datei einrichten
  DATA    Datei löschen
  DATA ----------------, Information, Quit,''''
  DATA SERVICE
  DATA - BLOCK 1 ----------------
  DATA   Menüpunkt 1a
  DATA   Menüpunkt 1b
  DATA   Menüpunkt 1c
  DATA ---- BLOCK 2 ------------
  DATA      Menüpunkt 2a
  DATA      Menüpunkt 2b
  DATA      Menüpunkt 2c
  DATA ------ BLOCK 3 ----------
  DATA         Menüpunkt 3a
  DATA         Menüpunkt 3b
  DATA         Menüpunkt 3c
  DATA ~~~~
  MENU m_punkt$()                 // Menü initialisieren
RETURN
PROCEDURE w_cls(wx%)
  COLOR wx%*3                     // Fensterfarbe
  DEFFILL wx%*7                   // Fensterfüllung
  PBOX 0,0,_X,_Y                       // Rechteck zeichnen
RETURN
PROCEDURE redraw(x%,y%,b%,h%)
  FOR I%=1 TO 4                   // vier Fenster
    WIN #I%                       // Fenster-Init
    CLIP OFFSET WIND_GET(4),WIND_GET(5)   // clipping setzen
    GETFIRST #I%,wx%,wy%,wb%,wh%       // erstes Rechteck holen
    WHILE wb% OR wh%                   // solange vorhanden ...
      IF RC_INTERSECT(x%,y%,b%,h%,wx%,wy%,wb%,wh%) //Überlappung??
        CLIP wx%,wy%,wb%,wh%      // Überlappungsrechteck clippen
        w_cls(I%)                 // Scheibenwischer an
      ENDIF
      GETNEXT wx%,wy%,wb%,wh%     // gfls. nächstes Rechteck holen
    WEND
  NEXT I%                         // nächstes Fenster
  WIN #WIND_GET(16)               // Fenster-Init wieder wie gehabt
RETURN
PROCEDURE gsbox(md%,st%,p%,qx%,qy%,qb%,qh%,zx%,zy%,zb%,zh%,b%,h%)
```




```
' Produziert eine 'GROW'- und/oder 'SHRINK'- und/oder 'MOVE'-Box
  '-----------------------------------------------------------
' Eine GROWBOX bzw. SHRINKBOX ist ein grafisches Rechteck, das
' sich - von einer Ursprungsgrösse ausgehend - dynamisch bis zu
' einer gewünschten Grösse vergrössert (grow) bzw. verkleinert
' (shrink). Eine MOVEBOX ist ein Rechteck mit konstanter Größe,
' das sich von einer bestimmten Bildschirmposition zu einer
' anderen bewegt (move).
  ' ----------------------------------------------------------
' md%  : Arbeitsmodus (4Bit-Vektor)
'        Bit 0 gesetzt (+1) = Quellbox wird 'geshrinkt'
'        Bit 1 gesetzt (+2) = 'Move'-Effekt wird ausgeführt
'        Bit 2 gesetzt (+4) = Zielbox wird 'gegrowt'
'        Bit 3 gesetzt (+8) = Quell- und Zielbox vertauschen
'
' st%  : gibt die Anzahl der Bewegungsschritte an (beliebig)
'
' p%   : gibt Verzögerung an (0-100, gfls. für schnelle 386er)
'
' qx%,qy%,qb%,qh% = Xpos, Ypos, Breite und Höhe der Quellbox
' zx%,zy%,zb%,zh% = Xpos, Ypos, Breite und Höhe der Zielbox
'
' b%,h%  : Breite und Höhe der Bewegungsbox (MOVE)
  ' ----------------------------------------------------------
LOCAL I%,j%,k%,l%,ls%,le%,x,y,xx,yy
DEFLINE -%1001001001001001    // idealer Linienstil
GRAPHMODE 3                   // XOR-Modus an
IF md% AND 8                  // Quell- und Zielbox tauschen
  SWAP qx%,zx%
  SWAP qy%,zy%
  SWAP qb%,zb%
  SWAP qh%,zh%
ENDIF
FOR k%=0 TO 2                 // je ein Durchgang für Quell-,
  '                           // Move- und Zielbox
  IF md% AND (2^k%)           // entsprechendes Bit gesetzt?
    IF k%=0 THEN ls%=0,le%=st%/2  // Marschrichtung
    IF k%=1 THEN ls%=0,le%=st%    // jeweils
    IF k%=2 THEN ls%=st%/2,le%=st% // einstellen
    FOR j%=0 TO 1                 // zwei Zeichen-Durchgänge
      FOR I%=ls% TO le%           // Schrittschleife
        IF k%=0                   // Quellbox ist dran?
          x=qx%+I%*((qb%/2)/st%)  // Koord. ...
          y=qy%+I%*((qh%/2)/st%)  // berechnen...
          xx=qx%+qb%-I%*((qb%/2)/st%)   // ...
          yy=qy%+qh%-I%*((qh%/2)/st%)   // ...
```

```
        ELSE IF k%=1                    // Movebox dran?
          x=qx%+qb%/2+I%*((zx%+zb%/2)-(qx%+qb%/2))/st%-b%/2
          y=qy%+qh%/2+I%*((zy%+zh%/2)-(qy%+qh%/2))/st%-h%/2
          xx=qx%+qb%/2+I%*((zx%+zb%/2)-(qx%+qb%/2))/st%+b%/2
          yy=qy%+qh%/2+I%*((zy%+zh%/2)-(qy%+qh%/2))/st%+h%/2
        ELSE                            // Zielbox dran?
          x=zx%+zb%/2-I%*((zb%/2)/st%)
          y=zy%+zh%/2-I%*((zh%/2)/st%)
          xx=zx%+zb%/2+I%*((zb%/2)/st%)
          yy=zy%+zh%/2+I%*((zh%/2)/st%)
        ENDIF
        DRAW x,y+(yy-y)/4 TO x,y TO x+(xx-x)/4,y // zeichnen...
        DRAW xx-(xx-x)/4,y TO xx,y TO xx,y+(yy-y)/4       // ...
        DRAW x+(xx-x)/4,yy TO x,yy TO x,yy-(yy-y)/4       // ...
        DRAW xx,yy-(yy-y)/4 TO xx,yy TO xx-(xx-x)/4,yy   // ...
        ' FOR l%=0 TO p%*100    // Brems-Schleife
        FOR l%=0 TO p%*100*intel% // Intelfakor nur in dieser Demo
        NEXT l%
      NEXT I%                   // nächsten Schritt
    NEXT j%                     // nächsten Zeichendurchgang
  ENDIF
  NEXT k%                       // nächste Box
  GRAPHMODE 1                   // Grafikmodus wieder normal
  DEFLINE 1                     // Nur für diese Demo
RETURN
PROCEDURE Path(p_str$,P_sgn$,VAR p_bk$,d_bk$,f_bk$)
    '-----------------------------------------------------------
  ' Untersucht einen String ab Stringende rückwärts auf
  ' das erste Vorkommen eines in einer Liste vorgegebenen
  ' Zeichens. Wird das (eines der) Suchzeichen gefunden,
  ' wird der String bei diesem Zeichen geteilt und die
  ' zwei Teile zurückgegeben. Als dritte Rückgabe erhält
  ' man dann das gefundene Zeichen. Wird es nicht gefunden,
  ' wird in der ersten Rückgabevariablen der gesamte Suchstring
  ' zurückgegeben und die beiden anderen Rückgabevariablen
  ' sind leer.
    '-----------------------------------------------------------
  ' P_str$ = zu untersuchender String
  ' P_sgn$ = String der das/die Trennzeichen enthält
  ' P_bk%  = Rückgabe-Stringvariable, die nach
  '          Abschlup den vorderen Stringteil enthält.
  ' D_bk%  = Rückgabe-Stringvariable, die nach
  '          Abschlup den hinteren Stringteil enthält.
  ' F_bk%  = Rückgabe-Stringvariable, die nach
  '          Abschlup das gefundene Zeichen enthält.
    '-----------------------------------------------------------
```




```
  LOCAL P_j%,P_fnd%
  IF LEN(p_str$)                   // ist was zum durchsuchen da?
    FOR P_j%=LEN(p_str$) DOWNTO 1// String rückwärts durchgehen
      P_fnd%=INSTR(P_sgn$,MID$(p_str$,P_j%,1)) //jedes Zeichen
      '                         // im Trennzeichenstring vergleichen
      EXIT IF P_fnd%               // Exit, falls fündig
    NEXT P_j%                      // nächstes Zeichen
    IF P_j%<1 OR LEN(p_str$)=1     // 'P_str$' nur 1 Zeichen lang
      '                            // oder nichts gefunden
      IF P_fnd%                    // im ersten Zeichen  gefunden
        p_bk$='''', d_bk$=''''     // keine Vorder- oder Hinterteil
        f_bk$=p_str$               // Suchzeichen ist 'P_str$'
      ELSE                         // kein Zeichen gefunden!
        p_bk$=p_str$               // 'P_str$' zurück
        d_bk$='''', f_bk$=''''     // sonst nichts
      ENDIF
    ELSE                           // Zeichen wurde gefunden
      p_bk$=LEFT$(p_str$,P_j%-1)      // Vorderteil zurück
      d_bk$=RIGHT$(p_str$,LEN(p_str$)-LEN(LEFT$(p_str$,P_j%)))
      '                            // Hinterteil zurück
      f_bk$=RIGHT$(LEFT$(p_str$,P_j%))// Trennzeichen zurück
    ENDIF
  ENDIF
RETURN
```

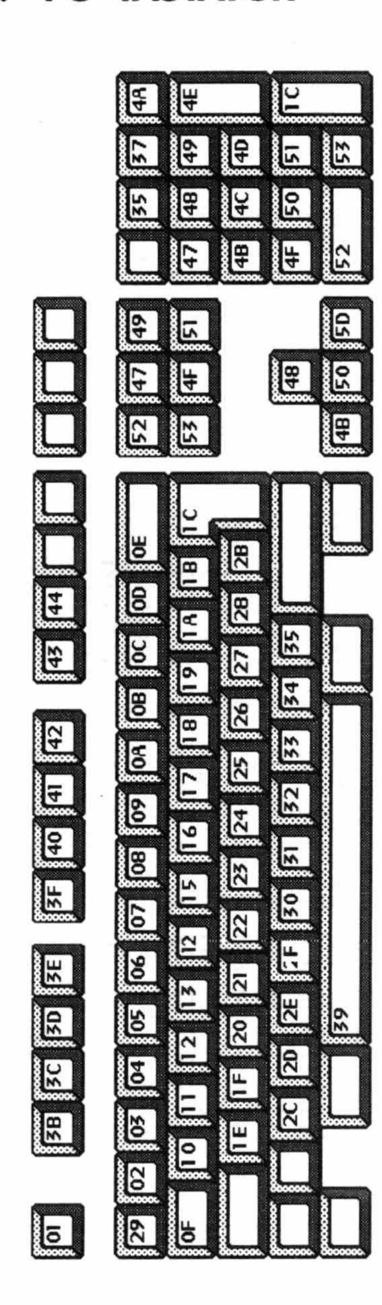
Notizen:

ANHANG B. ASCII-TABELLE

(IBM-Zeichensatz)

0	1 ☺	2 ☻	3 ♥	4 ♦	5 ♣	6 ♠	7 •	8 ◘	9 ○	10 ◙	11 ♂	12 ♀	13 ♪	14 ♫	15 ☼
16 ►	17 ◄	18 ↕	19 ‼	20 ¶	21 §	22 ▬	23 ↨	24 ↑	25 ↓	26 →	27 ←	28 ∟	29 ↔	30 ▲	31 ▼
32	33 !	34 "	35 #	36 $	37 %	38 &	39 '	40 (	41)	42 *	43 +	44 ,	45 -	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7	56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G	72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W	88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _
96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g	104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w	120 x	121 y	122 z	123 {	124 ¦	125 }	126 ~	127 ⌂
128 Ç	129 ü	130 é	131 â	132 ä	133 à	134 å	135 ç	136 ê	137 ë	138 è	139 ï	140 î	141 ì	142 Ä	143 Å
144 É	145 æ	146 Æ	147 ô	148 ö	149 ò	150 û	151 ù	152 ÿ	153 Ö	154 Ü	155 ¢	156 £	157 ¥	158 ₧	159 ƒ
160 á	161 í	162 ó	163 ú	164 ñ	165 Ñ	166 ª	167 º	168 ¿	169 ⌐	170 ¬	171 ½	172 ¼	173 ¡	174 «	175 »
176 ░	177 ▒	178 ▓	179 │	180 ┤	181 ╡	182 ╢	183 ╖	184 ╕	185 ╣	186 ║	187 ╗	188 ╝	189 ╜	190 ╛	191 ┐
192 └	193 ┴	194 ┬	195 ├	196 ─	197 ┼	198 ╞	199 ╟	200 ╚	201 ╔	202 ╩	203 ╦	204 ╠	205 ═	206 ╬	207 ╧
208 ╨	209 ╤	210 ╥	211 ╙	212 ╘	213 ╒	214 ╓	215 ╫	216 ╪	217 ┘	218 ┌	219 █	220 ▄	221 ▌	222 ▐	223 ▀
224 α	225 ß	226 Γ	227 π	228 Σ	229 σ	230 µ	231 τ	232 Φ	233 Θ	234 Ω	235 δ	236 ∞	237 φ	238 ε	239 ∩
240 ≡	241 ±	242 ≥	243 ≤	244 ⌠	245 ⌡	246 ÷	247 ≈	248 °	249 ∙	250 ·	251 √	252 ⁿ	253 ²	254 ■	255

ANHANG C. 'PC-TASTATUR'



Scan-Codes bei **KEYGET** und **KEYTEST**

ANHANG D. 'FEHLERLISTE'

-1	Falsche Funktionsnummer
-2	File nicht gefunden
-3	Pfad nicht gefunden
-4	Zuviele Dateien offen
-5	Keine Permission
-6	File Handle falsch
-7	MDB zerstört -- crashing
-8	Speicher voll
-9	Falsche Adresse
-10	Falsches Environment
-11	Format falsch
-12	Access Code falsch
-13	Invalid data
-14	Error -014
-15	Falsches Laufwerk
-16	RMDIR Fehler
-17	Kein Rename mit verschiedenen Laufwerken
-18	Datei Tabelle voll
-19	Argument falsch
-20	Argument Liste zu lang
-21	Exec Format falsch
-22	Cross-device link
-23	Error -023 -
-32	Error -032
-33	Math argument
-34	Bereichsfehler
-35	File existiret schon
-36	Error -036 -
-127	Error -127
-128	Stopped

0	Error 000
1	Division durch Null
2	Überlauf
3	Wertebereich
4	Negative Wurzel
5	SINGLE{} Überlauf
6	Error 006
7	Redimension
8	Feld nicht eindimensional
9	Feldindex zu groß
10	Dim zu groß
11	Falsche AnzahlDimensionen
12	GrafModeFail
13	String zu lang
14	Ausdruck zu komplex
15	EMS voll
16	EPopReDim
17	EMSInitFail
18	EPushNotDim
19	EMSAccFail
20	Feld zu klein
21	Falsche File#
22	File schon geöffnet
23	Open-Modus falsch
24	File nicht geöffnet
25	EMSempty
26	EMSTypeMismatch
27	EMSMemBlk2Large
28	EMSMemBlk2Small
29	PolyPtOvf
30	ArrTypeErr
31	Type Redefinition
32	Type undefiniert
33	Type passt nicht
34	Element Redefinition
35	Feld zu groß
36	FMSAccFail
37	EMSCfgAccFail
38	EMSNameNotFound
39	EMSMemBlk0
40	Procedure undefiniert
41	Parameter passt nicht
42	Zu viele Parameter
43	Zu wenig Parameter
44	MatrixErr

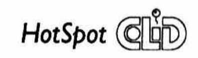
45	Bereichsfehler (Bound)
46	DATA-Ende erreicht
47	Data nicht numerisch
48	Falsche Fenster-Nummer
49	Fenster nicht geöffnet
50	WinNotTopped
51	C-Call Destroyed Register
52	Matrix nicht quadratisch
53	Matrixproduktnichtdefiniert
54	IllegalWndGSNum
55	IllegalWriteMd
56	NotInProcFor
57	LPT Fehler
58	COM Fehler
59	Illegale Laufwerksnummer
60	PUT-String-Fehler
61	Falscher SCREEN-Modus
62	Dateiende erreicht
63	Error 063...
87	Error 087
88	BSAVE/BLOAD Fehler
89	Fehlerhaftes GOTO
90	Random IO-Fehler
91	Error 091...
98	Error 098
99	Stack Überlauf
100	GFA-BASIC PC 4.31 (c) 1990 GFA Systemtechnik
101	Error 101...
121	Error 121
122	Proc/Func zu groß
123	FUNCTION undefiniert
124	ENDFUNC erreicht
125	Math Err 2
126	Load Error
127	Math Err 1

Notizen:

ANHANG E. 'THEMEN-ÜBERSICHT'






HotSpot






HotSpot



ANHANG F. 'SYNTAXLISTE'











ANHANG G. 'BEFEHLSLISTE'

Notizen:

ANHANG H. 'STICHWORT - INDEX'





*Das **PREMIUM**-Buch zum*

GFA BASIC

für MSDOS



GFA BASIC für MSDOS

Zu diesem Buch

Dieser **COLID - HotSpot** bietet eine komplette Beschreibung aller Befehle und Funktionen des GFA-BASIC unter MSDOS und liefert Ihnen kompaktes Wissen über diese hervorragende Programmiersprache. Er wurde in erster Linie als Schnelleinstieg und komprimierte Übersicht mit ausführlichem Begriffsindex, sowie Befehls-, Syntax- und Fehlertabellen konzipiert und ist so mit seiner geballten Information im handlichen Taschenformat jederzeit griffbereit.

Der Autor

Uwe Litzkendorf studierte Architektur, als er 1983 aus Neugier erst über den legendären COMMODORE C64 und dann den ATARI ST in die Computerei einstieg. Er war einer der ersten, die mit dem GFA BASIC im Frühjahr 1985 in Berührung kamen. Nach einer persönlichen Einweisung durch Frank Ostrowski - den Vater dieser hervorragenden Sprache - schrieb er Das große GFA - BASIC Buch´, das innerhalb kurzer Zeit zum Bestseller wurde. Mit einer Gesamtauflage seiner Bücher in der BRD, Frankreich, den USA und Holland von über 120.000 Exemplaren erreichte er mit seiner leicht verständlichen Sprache viele Hobby- und Profi-Programmierer am ATARI ST und dem COMMODORE AMIGA, die sein profundes Wissen zum Thema GFA-BASIC schätzen gelernt haben.



DM 59,80
öS 421,--
sFr 53,60

ISB N 3-9802925-0-9





SC 2